

École polytechnique de Louvain

An Ultra-Low-Power Embedded Convolutional Neural Network for Acoustic Monitoring of Forest Ecosystems

Author: **Thomas DELSART**

Supervisor: **David BOL**

Readers: **Christophe DE VLEESCHOUWER, Laurent JACQUES, Martin LEFEBVRE**

Academic year 2020–2021

Master [120] in Electrical Engineering

Abstract

For a few years now, artificial intelligence has brought significant improvements in the field of audio classification. However, complex audio recognition tasks often require powerful machine learning models that consume a substantial amount of energy to be trained and operated. One of the most important challenges these promising techniques have to solve is to be integrated inside embedded devices such as microcontrollers and Internet-of-Things edge devices. To this purpose, TinyML is a fast-emerging area of Deep Learning performing inference at extremely low power, in the mW range and below, thus enabling the deployment of millions of environmentally-friendly smart connected devices. One of the major applications is keyword spotting that allows always-on speech-based user interactions. Moreover, due to global warming, forest ecosystems audio monitoring has become more popular because it helps to fight deforestation, which is an essential factor in climate change. Furthermore, audio monitoring in forests also allows fighting animal poaching and positively impacts wildlife conservation by monitoring bird and mammal populations.

The focus of this Master thesis is to obtain an embedded ultra-low-power convolutional neural network (CNN) able to classify sounds from a forest environment among the animals, birds, chainsaws, and gunshots classes. The task is to accurately classify samples from a real-world crafted dataset on the Apollo3 Blue microcontroller and to obtain a power consumption below the mW range. The key question of this work is to know if it is possible to get an ultra-low-power high-accuracy convolutional neural network allowing to classify forest sounds. In that way, the proposed dataset, partly based on professional datasets from *Rainforest Connection* and the *Borror Laboratory of Bioacoustics*, is processed by changing the sample rate to 16 kHz and segmenting the recordings into 1-second audio slices to create the used dataset. An accuracy of 99.64% is obtained on this crafted dataset based on the MobileNetV1 architecture and by using the Mel-frequency cepstral coefficients optimized feature set as well as known audio augmentation techniques. Due to the power-consuming nature of this model, a shallow network based on depthwise separable convolutions and dense operations is designed. It reaches an accuracy of 93.52% and performs $273\times$ fewer MACC operations than the MobileNetV1 baseline model. A real-time implementation of the optimized model reaching an accuracy of 91.37% is then achieved with an 8-bit fixed-point on the Apollo3 Blue microcontroller, using the ARM CMSIS-NN library. Due to the use of 4-lanes parallel SIMD instructions and cache pre-fetching on the Flash accesses, the design demonstrates an inference time of 4.46 ms. By enabling the deep sleep mode of the microcontroller after the inference, the overall system consumes as little as $13.21\text{ }\mu\text{W}$ on average by performing one inference per second. It would therefore be possible to deploy thousands of these highly energy-efficient embedded CNNs in forests to detect deforestation and animal poaching as well as contributing to wildlife conservation.

Acknowledgments

This thesis has been written throughout my last year at UCLouvain. During this considerable work, I would like to thank the following people for their help, hints, and sources of motivation.

First of all, I would like to thank my supervisor, Professor David Bol, for his availability, support, and advice. I am also grateful for the opportunity he gave me to conduct this important work.

Secondly, I would like to thank Professor Christophe De Vleeschouwer, Martin Lefebvre, Adrian Kneip, and Julien Verecken for their constant supervision throughout the year. They proved their availability at any time and had precious hints. My thanks also go to Sami Gharib, who helped me with the CMSIS-NN implementation of the network.

Finally, I would like to sincerely thank Zephyr Gold and Anthony Harfield from the *Rainforest Connection* team as well as Kandace Glanville, former curatorial assistant of the *Borrór Laboratory of Bioacoustics* and Tamaki Yuri, curator of the *Borrór Laboratory of Bioacoustics*. This thesis would not have yielded the same results without their help in obtaining relevant audio datasets.

1	Introduction	22
2	Machine Learning & Deep Learning Background	25
2.1	Convolutional Neural Networks	26
2.1.1	Convolutional layer	27
2.1.2	Depthwise separable convolution layer	28
2.1.3	Pooling layer	29
2.1.4	Activation layer	30
2.1.5	Fully connected layer	31
2.2	Training process	31
2.2.1	Loss function	32
2.2.2	Backpropagation algorithm	33
2.2.3	Gradient Descent	34
2.3	Evaluation metrics	35
2.4	Computational complexity metric	37
2.5	Overfitting in neural networks and prevention	38

2.5.1	Data augmentation technique	38
2.5.1.1	Image-based techniques	39
2.5.1.2	Audio-based techniques	39
2.5.2	Regularization techniques	42
2.5.2.1	Complexity of the network	42
2.5.2.2	Weights regularization	42
2.5.2.3	Dropout	42
2.5.2.4	Batch normalization	43
2.5.2.5	Others	44
2.6	Audio Classification	44
2.6.1	Analog and digital sound	44
2.6.2	Audio classification chain	46
2.6.3	Audio Spectral Features	47
2.6.3.1	Spectrogram	47
2.6.3.2	Log-amplitude spectrogram	48
2.6.3.3	Mel spectrogram	48
2.6.3.4	MFCC	49
2.6.3.5	Other less known features	49
3	State-of-the-art in Sound Classification	50
3.1	Environmental Sound Classification	50
3.1.1	Datasets	50
3.1.2	Related works	52
3.2	Music Classification	53

3.2.1	Datasets	53
3.2.2	Related works	54
3.3	Speech Classification	56
3.3.1	Datasets	56
3.3.2	Related works	56
3.4	Bird Classification	58
3.4.1	Datasets	58
3.4.2	Related works	59
3.5	Summary	60
4	Dataset Setup	61
4.1	Gathering of the original dataset	61
4.2	Analysis of the original dataset	63
4.2.1	Sample rate	63
4.2.2	Number of audio channels	64
4.2.3	Duration of the audio recordings	65
4.2.4	Resolution of the audio recordings	65
4.2.5	Quality of the audio recordings	66
4.2.6	Proportion of useful signal with respect to the original one	66
4.2.7	Conclusions	67
4.3	Segmentation Technique	68
4.4	Compilation of the <i>Events</i> dataset	72
4.4.1	Animal dataset filtering	72
4.4.2	Optimal sampling rate choice	72

4.4.3	Resolution choice	76
4.4.4	Slice length choice	76
4.4.5	Segmentation of the recordings	77
4.5	Analysis of the <i>Events</i> dataset	78
4.6	Summary	79
5	Training of a baseline convolutional neural network on the <i>Events</i> dataset	80
5.1	Training setup	80
5.2	Baseline model	81
5.3	Experiments flow	83
5.3.1	Impact of the number of training samples on the classification	84
5.3.2	Choice of the feature type	85
5.3.2.1	Complexity	85
5.3.2.2	Number of pixels in the input slice	86
5.3.2.3	Validation accuracy	87
5.3.3	Choice of the data augmentation technique	88
5.3.4	Tuning of the training hyperparameters	92
5.3.5	Optimized baseline model	93
5.4	Summary	95
6	Model refinement for microcontroller deployment	97
6.1	Microcontroller selection	97
6.2	Impossibility of the baseline model on the Apollo3 Blue microcontroller	99
6.3	Experiments flow	99
6.3.1	Obtention of the model architecture	100

6.3.2	Extraction of the best feature type	102
6.3.3	Data augmentation technique	104
6.3.4	Hyperparameter tuning	105
6.3.5	Optimized refined model	107
6.4	Summary	109
7	Implementation of the convolutional neural network on an ultra-low-power micro-controller	110
7.1	Used Software	110
7.2	<i>CMSIS-NN</i> library	110
7.3	Conversion of a machine learning model into its <i>CMSIS-NN</i> implementation	113
7.3.1	Analyzing the <i>CMSIS-NN</i> functions	114
7.3.2	Analyzing the network	114
7.3.3	Dealing with the batch normalization layers	115
7.3.4	Reordering of the weights	117
7.3.5	Quantization of the weights	118
7.3.6	Quantization of the activations	120
7.3.7	Computation of the bias and output shifts	123
7.3.8	Generation of the <i>CMSIS-NN</i> implementation	124
7.4	Metrics of the network with <i>CMSIS-NN</i>	128
7.4.1	Test set accuracy	128
7.4.2	Runtime execution of the <i>CMSIS-NN</i> network	130
7.4.3	Power consumption of the device	132
7.5	Summary	135

8	Discussion	137
8.1	Limitation and larger context	137
8.2	Interesting issues	138
8.2.1	Sample rate review	138
8.2.2	Segmentation technique review	139
8.2.3	Model analysis	140
8.2.4	Microphone detection reliability	142
8.3	Comparison with the state-of-the-art	143
9	Improvements	146
9.1	Bird species classification	146
9.2	Wildfires detection	147
9.3	Multi-label classification	147
9.4	Refinement of the embedded device and library	147
10	Conclusion	148
	Appendices	151
A	Support information	152
A.1	Segmentation technique Python code	152
A.2	Implementation and illustration of the <code>step_decay</code> schedule on <i>Keras</i>	157
A.3	Implementation of the weighted categorical crossentropy loss on <i>Keras</i>	158
A.4	Hyperparameters of the seven studied features	159
A.5	<i>CMSIS-NN</i> implementation files	165
A.5.1	main.cpp	165

A.5.2	parameter.h	170
A.5.3	weights.h	172
A.5.4	input.h	173

LIST OF FIGURES

2.1	<i>Machine Learning</i> versus <i>Deep Learning</i> frameworks [23].	26
2.2	State-of-the-art accuracies on <i>ImageNet</i> [28].	26
2.3	Example of CNN architecture. The input image goes through a CNN containing convolutional, pooling, and fully connected layers. The activation layers are not represented except for the softmax activation layer. It allows converting the input of the layer into a probability to make the multi-class classification. This specific CNN can learn features from dog, bird, cat, and fish images from the training data due to the first part of the network. Then, the fully connected layers use this information to predict the class of the input image. [29].	27
2.4	The input image contains 3 channels (RGB) with 6 pixels in height and width. One convolutive filter is composed of three kernels of size 4×4 . For one output feature map, each kernel of the filter is convolved with its corresponding input image channel. The three processed channels are then added together to create one output feature map. If multiple filters are used in the convolution, as it is the case here, the same operation is performed with another filter (learned during the training). At the end of the operation for each filter, the feature maps are stacked together. Finally, the activation function and bias addition are performed pixel-wise on the output feature maps. [31].	28

2.5	Comparison between normal convolution (a) and depthwise separable convolution (b+c) [32].	29
2.6	Max pooling and average pooling operation. The kernel is of size 2×2 and travels the input feature map with a stride of 2. The input of size 4×4 becomes a feature map of size 2×2 [33].	30
2.7	Multiple examples of activation functions [35].	30
2.8	Fully connected layer. Input layer on the left and output layer on the right [36]. . . .	31
2.9	Cross-entropy loss function for binary classification [30].	33
2.10	Backpropagation algorithm. Step 1 represents the input data arriving in the network. Step 2 calculates the activation value of the neurons of the first hidden layer based on the current weights and the input data. In step 3, all activation values of the entire network have been computed based on Equation 2.3. Step 4 calculates the loss between the predicted output and the true one. Step 5 calculates the gradient of the output layer using the computed loss and backpropagates the gradients up to the first layer to calculate all weight gradients [39].	34
2.11	Confusion matrix for a binary classification task [41].	35
2.12	Confusion matrix for a multi-class classification task. TP, TN, FP, and FN need to be found for each class to calculate the metrics related to the specific class. This example makes the calculation for the 'Apple' class [41].	36
2.13	Overfitting phenomenon in a classification task. The training accuracy is 11% larger than the validation accuracy and the training loss is $19.5\times$ smaller than the validation loss. These two behaviors involve the overfitting of the model [44].	38
2.14	Time shift audio augmentation technique.	39
2.15	Noise injection audio augmentation technique.	40
2.16	Pitch shift audio augmentation technique.	40
2.17	Time stretch audio augmentation technique.	41
2.18	Mix-Up audio augmentation technique	41

2.19	Frequency & Time Masks audio augmentation technique [46].	42
2.20	Dropout principle. On the left is represented a simple neural network with 2 hidden layers. On the right is shown the same network after applying dropout. Several connections, controlled by the dropout rate of the layer, are removed to create a thinner network at training time [52].	43
2.21	Capture of an analog sound with a microphone, later converted into a digital representation with the ADC block [30].	44
2.22	Loss of information in the high frequencies due to the MP3 codec. The sound is a square wave with harmonics 1, 3, 5, ..., 19 [58].	46
2.23	Typical chain for classifying an audio sample [6].	46
2.24	Gunshot 2-seconds audio sample.	47
2.25	Gunshot spectrogram.	48
2.26	Gunshot log-amplitude spectrogram.	48
2.27	Mel scale [61]	49
2.28	Gunshot mel spectrogram.	49
2.29	Gunshot MFCC.	49
3.1	The 50 categories of the <i>ESC-50</i> dataset [64]. The categories underlined in red are part of the <i>ESC-10</i> dataset.	51
3.2	The ten categories of the <i>UrbanSound8K</i> dataset are: air conditioner, car horn, children playing, dog bark, drilling, engine idling, gunshot, jackhammer, siren, and street music. The number of samples per category is represented in the figure [65].	51
3.3	The <i>Broadcast Module</i> is composed of <i>Inception</i> blocks [93] stacked on top of each other and interconnected by dense connections [92].	55
3.4	Novel architecture used by S. Mittermaier & al. It is composed of a SincConv layer and depthwise separable blocks [111].	58

4.1	Topher White installing a Rainforest Guardian (sensor) high in the trees [15]	62
4.2	Diagram flow of <i>Rainforest Connection</i> mindset [133]	62
4.3	The sample rate is not constant. The sample rate of an audio from the original dataset can take one of the following values [8 ; 12 ; 16 ; 22.05 ; 32 ; 44.1 ; 48] [kHz].	64
4.4	The number of audio channels is not constant depending on the recording. The number of channels of an audio from the original dataset can take one of these values [1 ; 2]. .	64
4.5	Recordings duration histogram for each class. The duration of the audio samples depends a lot on the class.	65
4.6	Kernel density estimate of the proportion of useful signal with respect to the initial one for each class. The chainsaw dataset contains only interesting sounds, reason why the kernel density estimate of this class is not represented. The values -0.2 and 1.2 on the x-axis are due to the inner working of the KDE method.	67
4.7	Segmentation technique sequence of operations with the median blurring operation. .	69
4.8	Segmentation technique sequence of operations without the median blurring operation.	70
4.9	Block diagram of the segmentation technique.	71
4.10	Regions shaded in red are not taken (noise parts), while areas let in white are taken (signal parts). The author of the work chose 4% for the threshold since it gave good results. The audio file is a cat sound [143]	71
4.11	Logarithmic-amplitude spectrogram of a recording of each class at a sample rate of 48 kHz and 8 kHz.	73
4.12	Pipeline to find the optimal sample rate with our dataset. The red rectangles represent steps that are done on each temporal frame of the spectrogram.	75
4.13	Optimal sample rate : important steps of the pipeline.	75

4.14	Bird sound segmentation. For each subplot, the blue waveform is the original sound. The first subplot represents the mask output obtained by the segmentation technique. The other subplots represent the mask for the 1-second slices with different X thresholds. In these masks, a value set to 1 means that this recording part is taken into the <i>Events</i> dataset.	77
4.15	Proportion of the dataset among the four classes. The <i>Events</i> dataset is highly imbalanced with a large proportion of bird and chainsaw sounds.	78
5.1	Architecture of the <i>MobileNetV1</i> network [13]. Except for the first layer which is a normal convolution with a 3×3 kernel, Conv refers to a pointwise convolution layer followed by a batch normalization layer and a ReLU activation. Conv dw refers to a depthwise convolution layer followed by a batch normalization layer and a ReLU activation.	82
5.2	Diagram flow for all the experiments done with the baseline model.	83
5.3	This plot shows the positive impact of the number of training samples on the validation accuracy.	84
5.4	This figure is a violin plot and represents the time required to compute each variation of the seven studied features on a logarithmic scale. A bigger width involves more variations of the feature in the same y-coordinate. L-A Spectrogram means logarithmic-amplitude spectrogram. We note that the four features on the left are the best candidates as they need far less time to be computed.	85
5.5	Number of pixels in the input slice for each feature variation. L-A Spectrogram means logarithmic-amplitude spectrogram. We note that the two features on the right are the best candidates as they involve far fewer MACC operations due to their smaller shape.	87
5.6	Validation accuracy as a function of the number of MACC operations of the network. The Pareto front is also represented and its implementation is based on [161].	88
5.7	Confusion matrix on the validation accuracy of the baseline model before the data augmentation experiment, after the selection of the optimal feature type (MFCC_20).	89

5.8	Classification error on the validation set for different augmentation methods. Bars and whiskers indicate the mean and its 95% confidence interval computed from three repetitions of each experiment.	90
5.9	Difference of confusion matrices with and without data augmentation. Positive values indicate a positive change in confusion and negative values indicate a negative change in confusion.	91
5.10	Slice plot of the hyperparameters search space.	93
5.11	Accuracy and loss plots of the optimized baseline model.	94
5.12	Confusion matrix of the optimized baseline model on the test set of the <i>Events</i> dataset.	95
6.1	Diagram flow for all the experiments done to refine the baseline model.	99
6.2	LeNet5 architecture [24]	100
6.3	<i>Strided-DS 24</i> architecture [30]. The keyword 'F' designates the number of filters or feature maps in each layer, whereas the keyword 'D' is the stride size in each layer. Conv2d refers to a normal convolution layer, whereas CONV is a depthwise separable convolution layer. Padding is used in the architecture. A kernel size of (3,3) is used for each layer, and the dropout factor is 0.5.	101
6.4	The keyword 'F' designates the number of filters or feature maps in each layer, whereas the keyword 'D' is the stride size in each layer. CONV refers to a depthwise separable convolution layer. Padding is used in the architecture. A kernel size of (3,3) is used for each layer and the dropout factor is 0.5.	101
6.5	Each studied network is shown in a plot : model parameters versus the number of MACC operations with their specific validation accuracy. The green area corresponds to the requirements we fixed in Section 6.2. The Pareto front is represented in a solid red line.	102
6.6	10 variations of the MFCC feature found for the baseline model (<i>MFCC_0</i>) are studied in terms of the number of MACC operations, the complexity, and the validation accuracy.	103

6.7	Classification error on the validation set for different augmentation methods. Bars and whiskers indicate the mean and its 95% confidence interval computed from five repetitions of each experiment.	104
6.8	Difference of confusion matrices with and without data augmentation. Positive values indicate a positive change in confusion, and negative values indicate a negative change in confusion.	105
6.9	Search space for the hyperparameter tuning experiment. Feature maps Lx represents the number of feature maps in the depthwise separable convolution layer number x , taking one of these values $\{1, 2, 3, 4\}$	106
6.10	Hyperparameter tuning search space exploration of the refined model. Nber_filtersX represents the number of features map in layer X	107
6.11	Accuracy and loss plots of the optimized refined model.	108
6.12	Confusion matrix of the optimized baseline model on the test set of the <i>Events</i> dataset.108	
7.1	Comparison between the naive convolution operation and the optimized convolution using the <i>im2col</i> algorithm [184]. Note in this representation that the number of squares for each matrix is not always representative of the reality, whereas the noted dimensions are.	111
7.2	Overview of the <i>CMSIS-NN</i> kernel structure [14]. The NNFunctions part implements the different layers used in neural networks and these functions use utility functions from NNSupportFunctions	112
7.3	Overview of the step <i>Analyzing the CMSIS-NN functions</i> . Step 1 is the starting point and Step 2 the endpoint. Stop means that it is impossible to create the <i>CMSIS-NN</i> implementation of the network.	114
7.4	Overview of the step <i>Analyzing the network</i> . Step 2 is the starting point and Step 3 the endpoint.	115
7.5	Overview of the step <i>Dealing with the batch normalization layers</i> . Step 3 is the starting point and Step 4 the endpoint.	116

7.6	Overview of the step <i>Reordering of the weights</i> . Step 4 is the starting point and Step 5 the endpoint.	118
7.7	Original and quantized weights of the last pointwise convolution layer of our network.	119
7.8	Overview of the step <i>Quantization of the weights</i> . Step 5 is the starting point and Step 6 the endpoint.	120
7.9	Activations histograms of three different layers. The activations are floating-point values. The histograms bins scale at a power-of-two as it allows knowing the possible Q formats.	121
7.10	Pipeline to find the input and output Q formats of each layer in the <i>CMSIS-NN</i> implementation [195].	121
7.11	Overview of the step <i>Quantization of the activations</i> . Step 6 is the starting point and Step 7 the endpoint.	122
7.12	Overview of the step <i>Computation of the bias and output shifts</i> . Step 7 is the starting point and Step 8 the end point.	124
7.13	The different files used in the final <i>CMSIS-NN</i> implementation. The file in red is the most important one. It contains the application and the convolutional neural network implementation. The files in yellow are secondary files that are used by the main file.	125
7.14	Methodology to calculate the size of the scratch_buffer . Its total size is of 1600 bytes.	127
7.15	Overview of the step <i>Generation of the CMSIS-NN implementation</i> . Step 8 is the starting point and End the finalization of the complete process to convert a network into its <i>CMSIS-NN</i> implementation.	128
7.16	Confusion matrix on the test set.	129
7.17	Application running on the microcontroller to measure the power consumption of the latter.	133

7.18	Setup used to measure the power consumption of the Apollo3 Blue microcontroller. The output of the oscilloscope shows the active-deep sleep mode variations with a large part of the runtime in deep sleep mode (GPIO set to 0) due to the short inference time compared to the second we have at our disposal.	133
7.19	Power measurements results depending on the configuration of the microcontroller. Each bar has three colors that are the three parameters used in the configuration. . .	134
8.1	Overview of the design outside this Master thesis using our embedded software convolutional neural network. The green block is the always-on microphone designed by [217], the yellow block is our embedded convolutional neural network, and the red blocks are the remaining parts.	137
8.2	The scaling of the number of MACC operations of the optimized refined model with respect to the sample rate of the recordings demonstrates a behavior close to the linear one.	139
8.3	An example of sound event detection. The upper panel shows the input spectrogram, the center panel shows the ground truth, and the lower panel shows the prediction [224].	140
8.4	Three test samples from <i>ImageNet</i> systematically occluded with a grey square in the first column. The second column represents the probability of the CNN to output the correct class as a function of the occlusion position in the input image. The red color represents a high probability, whereas the blue color represents a low probability of correct class. The third column shows the most probable class to be outputted by the CNN with respect to the position of the grey square [226].	141
8.5	One test sample represented in the first column. The second column represents a map showing the discriminative parts of the image for a specific class. The last column proves that the GRAD-CAM technique provides similar results as occlusion sensitivity while being much cheaper to compute [227].	142

9.1	The overview of the two convolutional neural networks on top of each other. The first CNN on the left is the one created in this work, while the second on the right is the improvement allowing bird species classification. The picture on the right of the figure comes from [240].	146
A.1	Step_decay schedule used by the <i>LearningRateScheduler</i> Keras callback. The learning rate is divided by two every ten epochs in this example [246].	157

LIST OF TABLES

3.1	State-of-the-art convolutional neural networks for the <i>ESC-10</i> , <i>ESC-50</i> , and <i>Urban-Sound8K</i> datasets. We present the authors of the model and the paper in the first column. The feature types used by the model are mentioned in the second column. The three last columns are the accuracies [%] obtained on the benchmarks datasets. The work highlighted in orange has been implemented on a microcontroller.	53
3.2	State-of-the-art convolutional neural networks for the <i>GTZAN</i> and <i>Ballroom</i> datasets. The authors of the model and the paper are shown in the first column. The feature set used by the model is mentioned in the second column. The two last columns are the accuracies [%] obtained on the benchmarks datasets.	55
3.3	This table presents state-of-the-art convolutional neural networks for the <i>Google Speech Commands V1</i> and <i>Google Speech Commands V2</i> datasets with 12 words to be classified (a subset of the complete datasets). The authors of the model and the paper are shown in the first column. The feature set used by the model is mentioned in the second column. The two following columns are the accuracies [%] obtained on the benchmarks datasets. The last column is the number of parameters of the models in thousand units.	57
4.1	All audio datasets have been obtained from four organizations.	63
4.2	The different datasets do not have the same resolution.	65
4.3	The different datasets do not have the same audio quality.	66

4.4	Percentiles of the histograms [Hz].	76
5.1	General hyperparameters used for the experiments.	80
5.2	<i>MobileNetV1</i> width multiplier impact [13]. Reducing the width multiplier allows considerably decreasing the number of MACC operations as well as the number of parameters of the model. However, it comes at the cost of a loss of accuracy.	83
5.3	The number of MACC operations of the model decreases with the number of pixels in the input image. The values are obtained with the <i>keras-flops</i> Python package.	86
5.4	Validation accuracy of the baseline model before and after hyperparameter tuning . .	93
5.5	Metrics analysis of the baseline optimized model in terms of precision, recall, and F1-score on the test set.	95
6.1	Possible devices for our application with their specific global characteristics. All devices are based on a 32-bit RISC architecture. All cores are ARM Cortex-M based because we want to use <i>CMSIS-NN</i> . For the active current, when two values are given, it is the theoretical lowest one on the left and the biggest one on the right, depending on the chosen CPU frequency. The device in orange is a really good candidate but was not available at the time of this thesis.	98
6.2	More detailed characteristic of the chosen device.	98
6.3	Feature variations studied for the best feature set on the refined model.	103
6.4	Validation accuracy of the refined model before and after hyperparameter tuning. . .	106
6.5	Metrics analysis of the refined optimized model in terms of precision, recall, and F1-score on the test set.	108
7.1	Q formats obtained for the weights and biases of each layer of the network.	119

7.2	Q formats obtained for the activation values at the output of each layer for each quantization scheme. The input Q format of a layer is the output Q format of the previous layer. The activation layers are not shown in this table since their output Q format must be the same as their input Q format, meaning that it must be the same as the output Q format of the previous layer. The keyword Validation means that the whole validation set has been used to obtain the histograms, while Train refers to the entire training set.	122
7.3	Bias shifts obtained for each layer and each quantization scheme. The keyword Validation means that the whole validation set has been used to obtain the histograms, while Train refers to the entire training set.	123
7.4	Output shifts obtained for each layer and each quantization scheme. The keyword Validation means that the whole validation set has been used to obtain the histograms, while Train refers to the entire training set.	124
7.5	Test set accuracy of the <i>CMSIS-NN</i> network depending on the activations quantization scheme as well as the test set accuracy of the refined floating-point network.	129
7.6	Metrics analysis of the <i>CMSIS-NN</i> network compared to the refined floating-point network in terms of precision, recall and F1-score.	130
7.7	Runtime execution of the network in <i>CMSIS-NN</i> . The results have been obtained as the mean on 100 different inferences. The sum of the timing of each layer is not exactly equal to the total time due to the rounding at 3 decimals values.	131
7.8	Runtime execution of the network in <i>CMSIS-NN</i> when the flash cache is enabled. The cache characteristics are the following : direct-mapped, 128-bit line size, 512 entries (4 SRAMs active). The results have been obtained as the mean on 100 different inferences. The sum of the timing of each layer is not exactly equal to the total time due to the rounding at 3 decimals values. The sum of the proportions of the total runtime of each layer is not exactly equal to the total proportion due to the rounding at 2 decimals values.	132

8.1	Part 1 : state-of-the-art comparison for chainsaw and gunshot audio detection. The positive test recall corresponds to the recall for the chainsaw or gunshot class.	144
8.2	State-of-the-art comparison in terms of embedded CNN performance.	145
A.1	Spectrogram features studied in Section 5.3.2.	159
A.2	Log-amplitude spectrogram features studied in Section 5.3.2.	159
A.3	Mel spectrogram features studied in Section 5.3.2.	160
A.4	MFCC features studied in Section 5.3.2.	161
A.5	Cochleagram features studied in Section 5.3.2.	162
A.6	GFCC features studied in Section 5.3.2.	163
A.7	Scalogram features studied in Section 5.3.2.	164

For a few years now, artificial intelligence has brought significant improvements in the field of audio classification. However, complex audio recognition tasks often require powerful machine learning models that consume a substantial amount of energy to be trained and operated [1, 2, 3]. One of the most important challenges these promising techniques have to solve is to be integrated inside embedded devices such as microcontrollers and Internet-of-Things (IoT) edge devices [4]. To this purpose, TinyML [5] is a fast-emerging area of Deep Learning performing inference at extremely low power, in the mW range and below, thus enabling the deployment of millions of environmentally-friendly smart connected devices. One of the major applications is keyword spotting that allows always-on speech-based user interactions [6]. Another trend, due to global warming, is forest ecosystems audio monitoring that has become more popular because it helps to fight deforestation, which is an essential factor in climate change [7, 8, 9].

Forest ecosystems are essential to life on Earth. They provide air, water, food, and shelter to many living beings, one out of four people worldwide depending on the rainforests for their livelihoods [10]. They are home to more than half of the animal species worldwide, and they provide over 60% of anticancer drugs from the plants [10]. Furthermore, they allow us to fight global warming by storing massive amounts of carbon [11], and several studies have shown the benefits rainforests have on human mental health [12]. However, in recent years, illegal activities such as deforestation have endangered many living beings and accelerated global warming. Therefore, to keep the forest contributions, it is paramount to have environmentally-friendly IoT devices monitoring and protecting forest ecosystems.

In this Master thesis, we first gather a real-world audio dataset to classify audio samples between the animals, birds, chainsaws, and gunshots classes. A deeper analysis of the proposed dataset highlights the need to process it to be used efficiently by a convolutional neural network. To that purpose, we resample each recording to 16 kHz, found to be optimal with the proposed dataset, and we segment each recording into 1-second audio slices. Because this specific classification task has never been solved in the literature, we train a MobileNetV1 network [13] as a baseline model and optimize it with a set of multiple experiments in terms of accuracy and computational complexity. It yields a model achieving an accuracy of 99.64% with the Mel-frequency cepstral coefficients optimized feature set and some known audio augmentation techniques. Because of the power-consuming nature of this model, we perform another set of experiments to obtain a smaller convolutional neural network architecture in terms of memory footprint and computational complexity while preserving a high accuracy on the audio classification task. An embedded implementation of this refined model is then

achieved on the Apollo3 Blue microcontroller with the ARM CMSIS-NN library [14]. At the end of the process, the embedded convolutional neural network based on depthwise separable convolutions and dense operations allows classifying 1-second sounds among the animals, birds, chainsaws, and gunshots classes with a latency of 4.46 ms, a memory usage of 5 kB, an accuracy of 91.37%, and a power consumption of 13.21 μ W.

The objectives of this work are numerous. It first attempts to improve our world with artificial intelligence. To that purpose, this work allows us to fight deforestation, which plays a significant role in global warming and also destroys the lives of animals and poorer communities [15]. Furthermore, it allows sensing gunshot sounds to reduce the animal poaching activities worldwide, threatening many species with extinction [16]. Then, by listening to birds and animals, it enables censuses in specific areas of the world, therefore improving wildfire conservation [17]. Moreover, this work improves the quality of the monitoring in forest ecosystems, previously done by humans [18].

The work is structured as follows :

- **Chapter 2** describes the background in supervised machine learning and deep learning. First, this chapter focuses on the convolutional neural network architecture. Then, we address the training process of a neural network, the evaluation and computational complexity metrics, as well as the possible problems that can be encountered and the solutions to prevent or mitigate them. After that, we discuss the audio classification task in detail. First, a physical analysis of the sound is performed, and the different audio properties are mentioned. Finally, the typical audio classification pipeline and the features of the audio samples to provide to a convolutional neural network are given.
- **Chapter 3** analyzes benchmark datasets used for various audio classification tasks as well as their state-of-the-art convolutional neural networks.
- **Chapter 4** presents the gathering of the proposed audio dataset for the classification task targeted by this work. Then, we analyze and process the dataset to obtain a relevant audio dataset to work with, further referred to as the *Events* dataset.
- **Chapter 5** targets the training of a CNN to solve the *Events* classification task. The MobileNetV1 architecture is chosen to achieve this task based on its state-of-the-art performance as well as its primary use in embedded devices. Furthermore, the training setup and the structure of the baseline model are presented. A set of experiments is then conducted to obtain the highest accuracy on the classification task while minimizing the number of MACC operations of the baseline model. Among these experiments, the type of feature, the data augmentation techniques, and the training hyperparameters are optimized.
- **Chapter 6** presents the Apollo3 Blue microcontroller used in this work and demonstrates that the baseline model cannot be fitted on this hardware platform. Therefore, this chapter aims to find a smaller architecture fitting the microcontroller requirements and reaching the highest possible accuracy on the classification task. To that purpose, we perform another set of experiments, including the architecture choice, a refinement of the feature type as well as the optimization of the data augmentation and hyperparameter tuning techniques. The obtained model is then used for the remainder of this work.
- **Chapter 7** introduces the CMSIS-NN library used to embed the optimized model on the Apollo3 Blue microcontroller and describes its advantages and working principles. The chapter then explains in detail how to convert a convolutional neural network from the original framework

into a CMSIS-NN implementation running on the microcontroller. Interesting metrics, including the test set accuracy of the embedded model, the runtime execution, and the power consumption of the entire device, are then presented. The latter has been obtained based on experimental measures on the microcontroller.

- **Chapter 8** gives the limitation of this work and discusses interesting issues. Then, we compare the final result of this Master thesis with the state-of-the-art.
- **Chapter 9** mentions interesting perspectives that could be reused later to improve this work.
- **Chapter 10** summarizes this work.

CHAPTER 2

MACHINE LEARNING & DEEP LEARNING BACKGROUND

Machine Learning and *Deep Learning* are two fields related to the *Artificial Intelligence* area. Their purpose is to create models trained on data and that can drive their own decisions. *Machine Learning* algorithms can be classified into 3 distinct categories :

1. Supervised learning : it is the most used type nowadays [19]. It consists of feeding pairs of input and output data to the model. The algorithm processes each pair and changes its parameters accordingly to mimic the known result. At the end of the process, it can make a decision on unseen related data. Two branches of algorithms exist in this category :
 - Regression tasks : the aim is to build a model that outputs a real number based on input features. Such an example is the *Boston house price prediction* task [20], in which the goal is to predict the price of a house based on a set of feature describing houses in Boston ;
 - Classification tasks : the objective is to create an algorithm able to classify an input into different classes. *ImageNet* classification task [21] is an example for which an input image must be classified into one of 1000 different classes.
2. Unsupervised learning : the input data given to the model does not contain labeled outputs. The purpose is to learn an interesting representation of the data. Clustering algorithms are such an example of this category. The aim is to learn features from the input data to cluster data samples in different groups.
3. Reinforcement learning : it consists of an agent (model) learning, by maximizing a reward, to perform actions in its environment. Basically, each action is rewarded if it is a good one or punished otherwise. This category knows significant improvements for a few years in games [22] ;

This Master thesis focuses on supervised learning audio classification algorithms. On the one hand, *Machine Learning* models need hand-crafted features as input to learn and make their predictions. Thus, there is a need for a feature extraction step, as shown in Figure 2.1, in which relevant information about the input data is retrieved and used in the network. On the other hand, *Deep Learning* models are more autonomous as they do not need this feature extraction block. When being given a lot of training samples, they are able to learn which specific information from the input samples to keep to drive the prediction. Nowadays, deep neural networks are state-of-the-art techniques in a wide range of applications from computer vision to natural language processing.

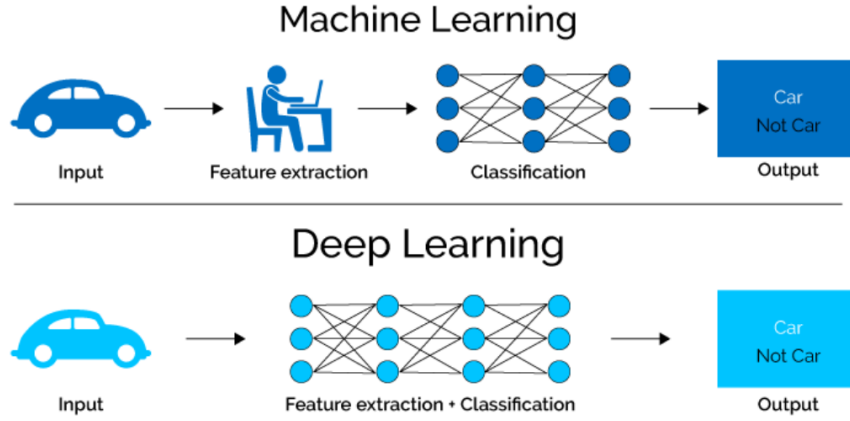


Figure 2.1: *Machine Learning* versus *Deep Learning* frameworks [23].

Firstly, this chapter aims to present in detail the convolutional neural network architecture, which is one of the most well-known *Deep Learning* supervised learning models. Then, the training process of neural networks, in general, is addressed as well as the important evaluation and complexity metrics used in classification tasks. Next, a section is dedicated to the explanation of the *overfitting* phenomenon and the ways to mitigate it. Finally, we discuss in detail the audio classification task and the typical processing chain for it. It includes an explanation of spectral features directly extracted from the raw waveforms.

2.1 Convolutional Neural Networks

Convolutional neural networks, also referred to as CNNs, are a type of neural network specialized in image data tasks. The architecture was used for one of the first times in 1998 by Yann LeCun for handwritten digit recognition [24]. They are now popular due to their state-of-the-art performance in a wide range of applications such as person detection [25], image classification [26], audio classification [2], image segmentation [27], etc. Figure 2.2 represents the evolution of the classification error on the *ImageNet* benchmark. We notice the out blowing accuracies that CNNs (on the right of the blue line) can yield on such a difficult task.

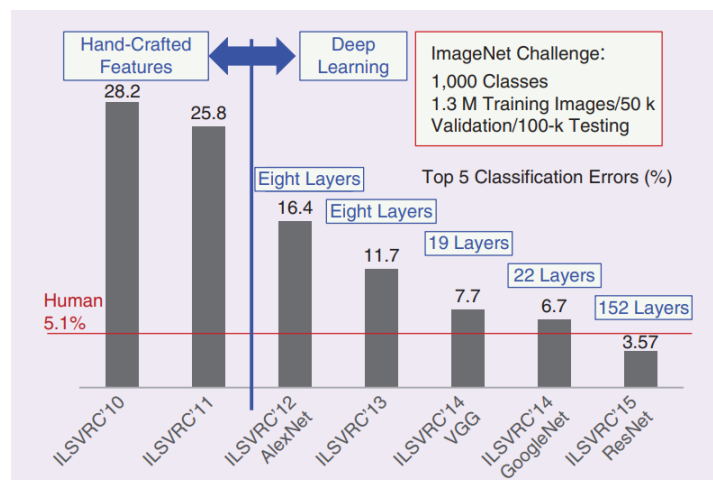


Figure 2.2: State-of-the-art accuracies on *ImageNet* [28].

A CNN architecture is made of convolutional layers to extract high-level features from the data and fully connected layers to make the classification based on the learned features. It can also contain pooling layers in the feature extraction part, which reduces the dimensionality of the feature maps. Each layer, regardless of its type, is followed by a non-linear activation layer. A typical CNN architecture is represented in Figure 2.3. Each layer is analyzed in the remainder of this section.

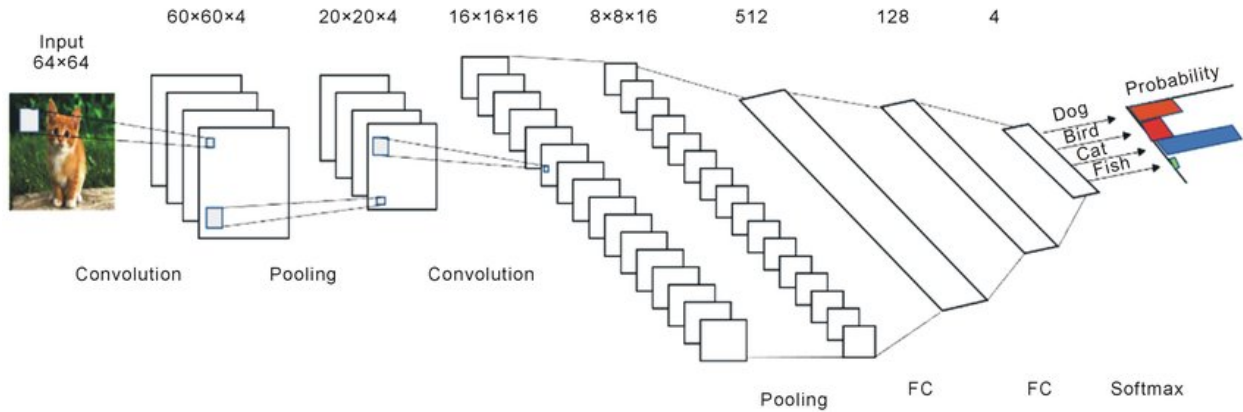


Figure 2.3: Example of CNN architecture. The input image goes through a CNN containing convolutional, pooling, and fully connected layers. The activation layers are not represented except for the softmax activation layer. It allows converting the input of the layer into a probability to make the multi-class classification. This specific CNN can learn features from dog, bird, cat, and fish images from the training data due to the first part of the network. Then, the fully connected layers use this information to predict the class of the input image. [29].

2.1.1 Convolutional layer

This layer takes as input an image or feature map. "It can exploit the spatial relationship in the data" [30]. Convolution layers work with 3 dimensions data : the height and width of the image and the number of channels of the latter. Typically, a color image contains RGB components and thus three channels. However, a grayscale image has only 1 channel. The convolution operation works as follows : the input image is convolved with a given set of filters, learned during the training. A filter is composed of a set of kernels for which the number of kernels is equal to the number of channels in the input image. Each kernel works on its corresponding channel of the input image. The kernel is swept across the image. For each slide, the input image pixels included in the kernel slide are multiplied by their corresponding weight from the kernel. These summations are then added together to output a single value. A convolutional layer has in practice several important parameters :

- The stride : it represents the amount of pixels for which the kernel is shifted for the convolution operation. The stride parameter is present in the width and height dimensions and is equal to 1 by default. However, it is common to increase this value to 2 to downsample the input feature map and reduce the number of operations ;
- The kernel size : it represents the size of the convolution kernels. Usually, we use square kernels. A bigger kernel size involves fewer operations for the convolution operation since it considers more pixels at a time ;
- The padding : it allows to zero pad the input feature map, such that the output feature map has the same size as the input feature map after the convolution operation ;
- The number of filters : it represents the number of feature maps that are outputted for a convolution operation with the input image. Each filter is convolved with the input image to

give an output feature map. Increasing the number of filters allows learning different features in the image at a given scale. For instance, a filter could be responsible for detecting vertical edges in an image, whereas another could blur the image.

Let us wrap up this layer : the convolution operation is represented in Figure 2.4.

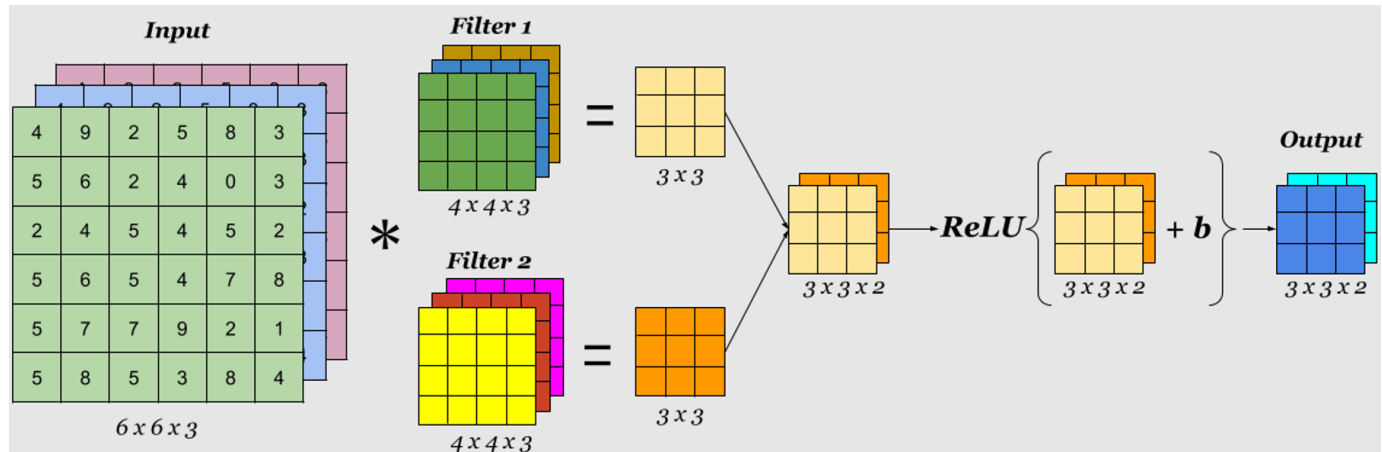


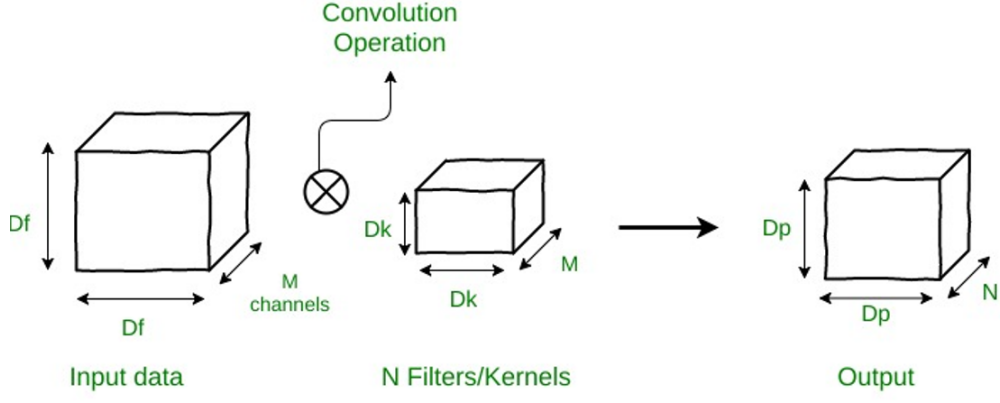
Figure 2.4: The input image contains 3 channels (RGB) with 6 pixels in height and width. One convolutive filter is composed of three kernels of size 4×4 . For one output feature map, each kernel of the filter is convolved with its corresponding input image channel. The three processed channels are then added together to create one output feature map. If multiple filters are used in the convolution, as it is the case here, the same operation is performed with another filter (learned during the training). At the end of the operation for each filter, the feature maps are stacked together. Finally, the activation function and bias addition are performed pixel-wise on the output feature maps. [31].

2.1.2 Depthwise separable convolution layer

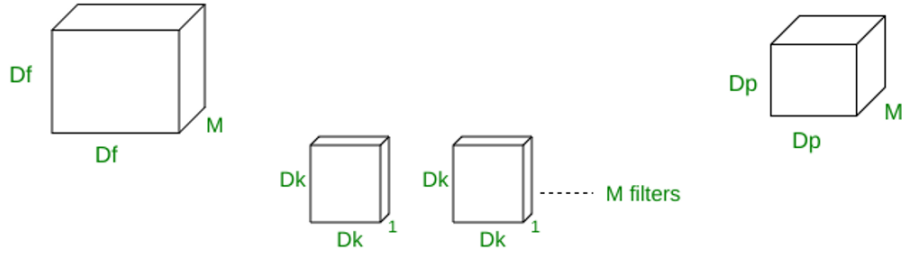
This is a variant of the regular convolution layer proposed in Section 2.1.1. It performs a convolution by splitting the operation into two steps :

1. Depthwise (spatial) convolution : the convolution operation is performed on one single channel at a time. The number of feature maps of this layer is thus equal to the number of channels of the input image ;
2. Pointwise convolution : this is a convolution with a kernel of size 1×1 directly applied to the output of the depthwise convolution. It contains N filters, with N the number of output feature maps.

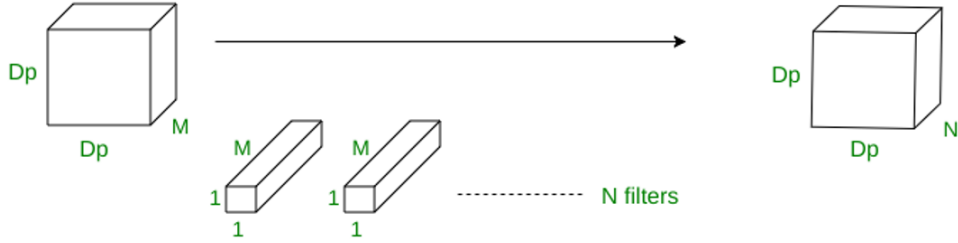
The comparison between a regular convolution operation and a depthwise separable convolution is shown in the following figure. Note that the advantage of the latter is that it performs much fewer operations compared to a standard convolution. This is explained in detail in Section 2.4.



(a) Normal convolution.



(b) Depthwise convolution.



(c) Pointwise convolution.

Figure 2.5: Comparison between normal convolution (a) and depthwise separable convolution (b+c) [32].

2.1.3 Pooling layer

A pooling layer takes as input an image or feature map and outputs a downsampled version of the input feature map. It allows reducing the size of the image, which forces the model to learn more complex features. Moreover, by downsampling the image, it involves fewer computations and parameters compared to a convolution layer. Pooling layers work with a fixed kernel size that is swept across the image. For each slide, a computation is performed on the pixels of the input image included in the kernel slide. The computation depends on the type of pooling layer. We distinguish the *Max pooling layer* and the *Average pooling layer*. The first one takes the maximum of the input image pixels included in the kernel slide, while the second type takes an average value of all the pixels contained in the kernel slide. The two processes are represented in Figure 2.6.

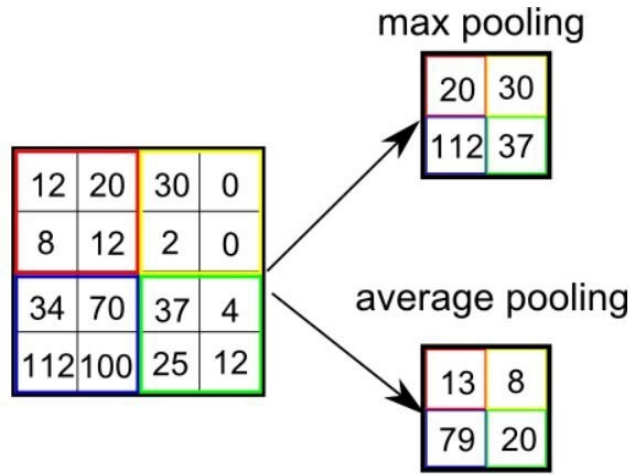


Figure 2.6: Max pooling and average pooling operation. The kernel is of size 2×2 and travels the input feature map with a stride of 2. The input of size 4×4 becomes a feature map of size 2×2 [33].

2.1.4 Activation layer

An activation layer is a function that takes neurons as input and performs a non-linear function on each of them. It allows expressing non-linear relationships between input and output, representing more closely real-world applications. Figure 2.7 shows many different activation functions. Among them, the ReLU activation function is a general-purpose function, probably the most known as it is used a lot in neural networks to prevent the vanishing gradient problem [34]. Softmax and sigmoid activation functions are often used in the output stage of a network for classification tasks. They allow converting the input data to a sort of probability range in $[0 ; 1]$. Softmax is used for multi-class classification, while sigmoid is used for binary classification. To get the inferred class from the continuous probability values obtained by one of the two activation layers, we take the class that contains the largest value.

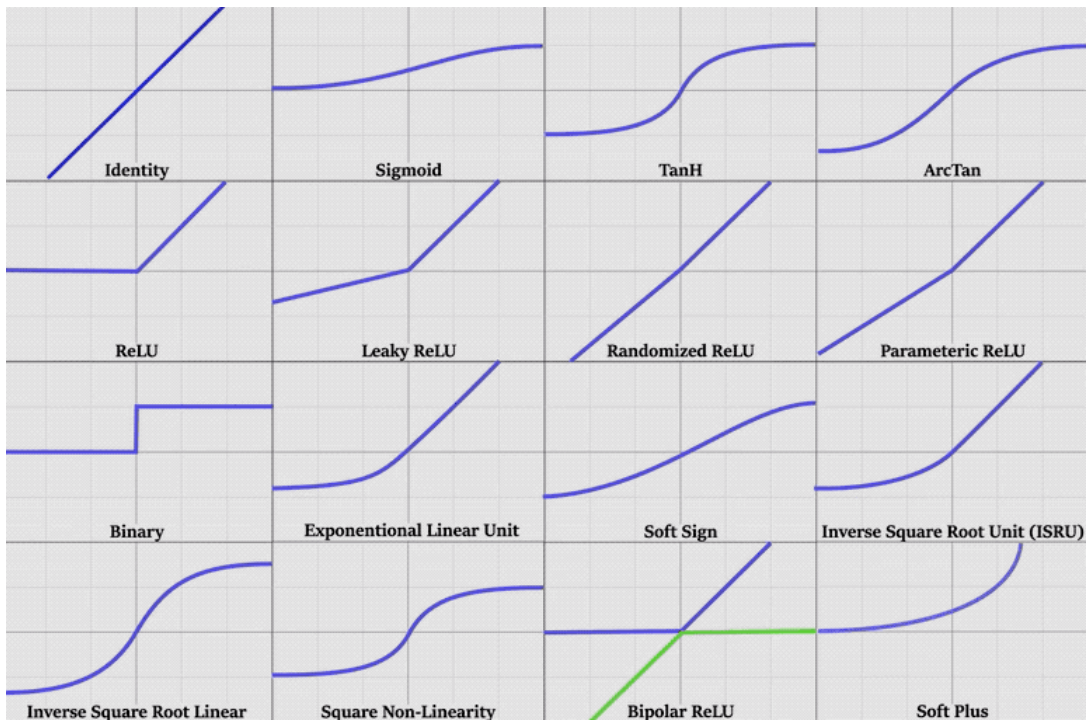


Figure 2.7: Multiple examples of activation functions [35].

2.1.5 Fully connected layer

This kind of layer has a set of neurons as input and another one as output. Each neuron value in the output is obtained by performing a weighted sum over every input, followed by the addition of the bias and the activation function. In other words, every neuron in the input layer is connected to every neuron in the output layer. The layer operation can be seen as a matrix product followed by the activation function :

$$Y = f(W^T X + b) \quad (2.1)$$

where $\mathbf{Y}$ is the 1D vector of output neurons, $\mathbf{X}$ is the 1D vector of input neurons, b is the bias of the layer, and $\mathbf{W}$ is a matrix representing the weights of the layer. There are n weights per input neuron if n is the number of output neurons.

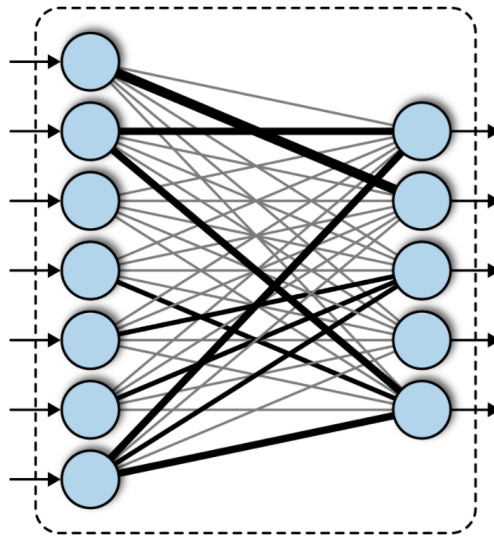


Figure 2.8: Fully connected layer. Input layer on the left and output layer on the right [36].

2.2 Training process

Training a neural network means modifying its parameters. The weights of a neural network can be initialized according to several schemes but none of them fits the training data initially. Therefore, we need to **train** the network such that it learns from the training data and is able to make predictions on similar unseen samples. Typical initialization schemes based on [37] are mentioned below :

- Random initialization
- Kaiming He initialization
- Xavier initialization

A good weights initialization scheme is important for the network convergence. [37] shows that setting all parameters to the same value involves the neurons of the network to have the same update, leading to a non-learning behavior of the network. Kaiming He initialization is a good choice with ReLU activation layers while Xavier initialization is best with Hyperbolic Tangent activation functions. Finally, random initialization works but yields in practice a bigger loss function at the end of the training [37].

To train a neural network efficiently, an important amount of data is typically needed. This large amount of data is called a dataset and it plays a major role in deep learning as it represents the entire

information learned and evaluated by the model. A diversified database, for which the properties of the data are the same, allows the model to learn more distinct features and to become more robust. Typically, a dataset is divided into three subsets for which the proportions depend on each work : the training, validation, and test sets. The first one is used to train the neural network with the backpropagation algorithm that we explain later in this section. The second set provides an unbiased evaluation for the trained model. Concretely, it allows during the training process to keep track of the generalization of the model. Finally, the test set contains unseen data by the model given to the trained model to check its generalization on new data samples. Therefore, it provides an evaluation of the fit model on new unseen samples.

To train a neural network, the training samples must be labeled. We also need to introduce the notion of the loss function, which is a metric allowing us to estimate the similarity between the output of the network and the true label from the training samples. With that knowledge, the backpropagation algorithm and the optimization process can be explained.

2.2.1 Loss function

The loss function is an essential metric to train a neural network. It allows measuring the resemblance between the predicted output of the neural network and the true one. It is used in the backpropagation algorithm to modify the parameters of the model such that it better fits the training data. Typically, in binary or multi-class classification, the **cross-entropy** loss function is used. Its equation is given here :

$$\mathcal{L} = - \sum_{c=1}^M (y_c \times \ln \hat{y}_c) \quad (2.2)$$

where $\mathbf{M}$ is the number of output classes in the classification task, $\mathbf{y}_c$ is the true output obtained from the training sample for the class $\mathbf{c}$, $\ln$ is the neperian logarithm, and $\hat{\mathbf{y}}_c$ is the output predicted by the neural network for the class $\mathbf{c}$ (the output of the softmax layer for the class $\mathbf{c}$). Note that $\mathbf{y}$ and $\hat{\mathbf{y}}$ are one-hot encoded vectors. It means that they have the same number of elements as the number of classes ($\mathbf{M}$). Each element of the vector represents one class. Therefore, as mentioned, $\mathbf{y}_c$ is the true output for the class $\mathbf{c}$.

The cross-entropy loss for binary classification is shown in Figure 2.9. Let us take a representative example: a machine learning model is built to classify an image between the cat and dog categories. At the end of the softmax layer, the network outputs the prediction $\hat{\mathbf{y}}_c = [0.1, 0.9]$, while the true output is $\mathbf{y}_c = [1, 0]$. Therefore, the network makes a wrong prediction with a high "probability" (90%). The cross-entropy loss function is thus $\mathcal{L} = -\ln 0.1 = 2.302$. The loss is big and goes towards infinity when the probability of correct classification diverges from the true label. The loss job is thus to penalize heavily wrong predictions during the training process by minimizing them. If the prediction of the network was $\hat{\mathbf{y}}_c = [0.55, 0.45]$, the loss function would be $\mathcal{L} = -\ln 0.55 = 0.597$. The prediction is still false but the probability of correct classification is higher, yielding a lower loss value.

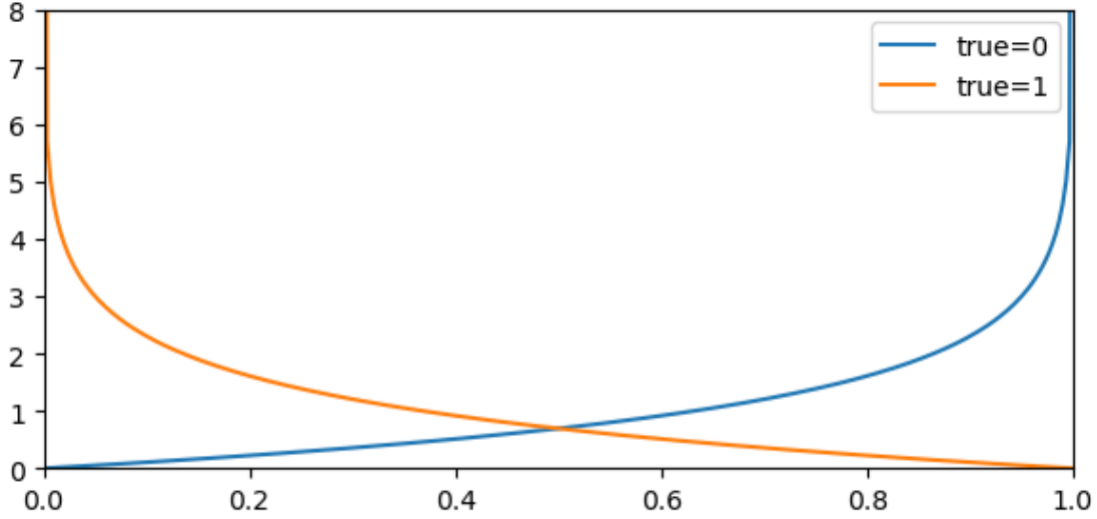


Figure 2.9: Cross-entropy loss function for binary classification [30].

2.2.2 Backpropagation algorithm

The training of a neural network uses the loss function introduced in the previous section in an algorithm known as the **Backpropagation** algorithm. This is the first ingredient to train a neural network. Its principle is explained below for a multi-layer perceptron network, made only of fully connected layers, based on [35]. A deep explanation about backpropagation in CNNs, which follows the same concept, can be found here [38]. Backpropagation allows computing the gradient of each weight of the neural network that is later used in the optimization process to modify the weights.

The backpropagation algorithm is done in two main steps :

- **The forward pass** : it allows computing the activation value of each neuron in the network up to the output layer (softmax layer in classification tasks), based on the current weight values and the current training sample. Mathematically, each activation is computed as follows :

$$a^{(l)} = \sigma(Wa^{(l-1)} + b) \quad (2.3)$$

where $a^{(l)}$ is the activation of the neuron in layer **l**, W is the matrix of weights of the layer **l-1**, $a^{(l-1)}$ is the activation of the neuron in layer **l-1**, and b is the bias of the layer **l-1**.

Having access to the output neuron(s) of the neural network, we are able to calculate the loss function presented in Section 2.2.1 which measures the similarity between the prediction of the network and the true label.

- **The backward pass** : We compute the gradient of each weight from each layer of the neural network. Gradients are basically small updates to each weight that represent the proportion and direction of output variation with small changes in the input. To that purpose, we begin by computing the gradient of the weight of the output layer, and we backpropagate this result up to the first layer. The equations needed to compute each gradient are proposed here [35]. They are based on the mathematical concepts of *partial derivatives* and *chain-rule* and they use the loss function as we want to minimize it.

The concept of backpropagation is represented visually in Figure 2.10. When the gradient of each weight in the network is known, the final step is the optimization process explained in the next subsection.

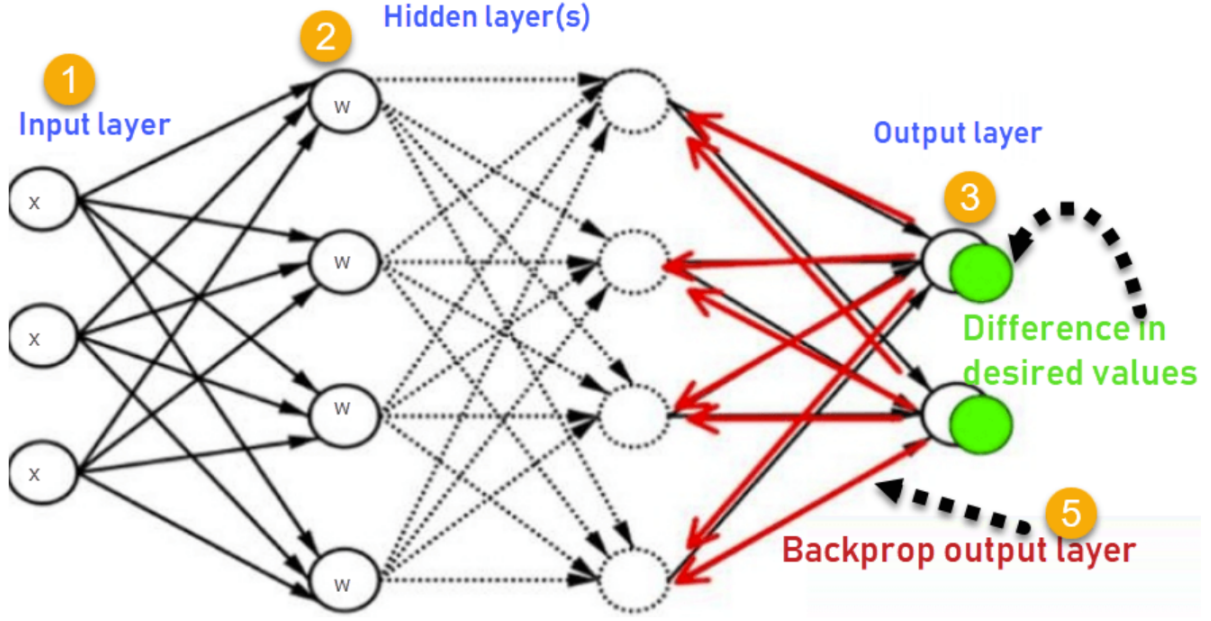


Figure 2.10: Backpropagation algorithm. Step 1 represents the input data arriving in the network. Step 2 calculates the activation value of the neurons of the first hidden layer based on the current weights and the input data. In step 3, all activation values of the entire network have been computed based on Equation 2.3. Step 4 calculates the loss between the predicted output and the true one. Step 5 calculates the gradient of the output layer using the computed loss and backpropagates the gradients up to the first layer to calculate all weight gradients [39].

2.2.3 Gradient Descent

This is the last ingredient allowing to train a neural network. It is an iterative optimization process that uses the computed gradients from the backpropagation algorithm and modifies each weight and bias of the network accordingly. Mathematically :

$$w^{(l)} = w^{(l)} - \eta \times \frac{\partial \mathcal{C}}{\partial w^{(l)}} \quad (2.4)$$

$$b^{(l)} = b^{(l)} - \eta \times \frac{\partial \mathcal{C}}{\partial b^{(l)}} \quad (2.5)$$

where $w^{(l)}$ is a weight in layer l , $b^{(l)}$ is a bias in layer l , $\mathcal{C}$ is the cost function, η is the learning rate, and $\frac{\partial \mathcal{C}}{\partial w^{(l)}}$ is the gradient of the cost function with respect to one weight in layer l . The learning rate is a hyperparameter, that is a parameter set before the learning process and which is therefore not learned. Its value is extremely important in the training process, typically in the range $[10^{-6}, 10^{-2}]$. A too big learning rate can lead to a non-convergence of the network whereas a too small learning rate can block the network in a local minimum [40].

In practice, the gradient descent algorithm is not used because the cost function is the mean of the loss functions evaluated on each training sample : $\mathcal{C} = \frac{1}{N} \times \sum_{i=0}^N \mathcal{L}_i$. It leads to memory and computation problems since each activation of the entire neural network has to be saved for each training sample. Moreover, each forward and backward pass are done on the complete training dataset, yielding an expensive computation. Therefore, to fix this problem, the *mini-batch gradient descent* algorithm is used since it is a good compromise as it approximates the classical gradient descent algorithm. This new technique requires a new hyperparameter, called the *batch size* (BS). It refers to the number of samples that are randomly taken from the training set for each weight update. With this algorithm,

the cost function becomes $\mathcal{C} = \frac{1}{BS} \times \sum_{i=0}^{BS} \mathcal{L}_i$ where BS is the batch size. This hyperparameter in practice must be high enough to provide a good estimate of the gradient descent algorithm and small enough to fit into memory.

In practice, during the training process of a neural network, the training dataset is presented multiple times to the network. A complete pass of the training dataset is called an epoch. We can now wrap up the complete training process of a neural network using the mini-batch gradient descent algorithm :

1. Take randomly a batch of data with a number of samples equals to the batch size ;
2. Perform the forward pass, compute the activation of each neuron for each sample and compute the cost function ;
3. Backpropagate the errors to compute the gradient of the weights in the complete network ;
4. Update the weights following Equation 2.4 and 2.5 ;
5. Go to 1) until all batches of all epochs are processed.

Finally, it is important to mention that many optimization algorithms are used nowadays and the mini-batch gradient descent algorithm is just one of them.

2.3 Evaluation metrics

Evaluating the performance of a deep learning model is essential as we want to have models that perform well on unseen data. The evaluation process is done on the validation or test set after the training, when the network weights are fixed. Multiple evaluation metrics exist in classification tasks but we only focus on the ones used in this work. The equations and principles are explained for binary classification. To that purpose, we introduce the notion of **True Positive**, **True Negative**, **False Positive**, and **False Negative** samples :

- A **True Positive (TP)** sample is correctly predicted positively by the network ;
- A **True Negative (TN)** sample is correctly predicted negatively by the network ;
- A **False Positive (FP)** sample is predicted positively by the network while it is negative ;
- A **False Negative (FN)** sample is predicted negatively by the network while it is positive ;

These four categories of samples can be represented in a confusion matrix shown in Figure 2.11. It is an important measure in classification tasks that allows seeing which class(es) are misclassified or correctly classified and their respective proportions.

		True Class	
		Positive	Negative
Predicted Class	Positive	TP	FP
	Negative	FN	TN

Figure 2.11: Confusion matrix for a binary classification task [41].

The other metrics of interest are the :

- **Accuracy** : it measures the fraction of the total number of correctly classified samples. It is calculated as $\frac{TP+TN}{TP+TN+FP+FN}$;
- **Recall** : it measures the fraction of positive samples correctly predicted by the neural network. It is calculated as $\frac{TP}{TP+FN}$;
- **Precision** : it measures the fraction of positive predictions made by the neural network with respect to the actual positive class. It is calculated as $\frac{TP}{TP+FP}$;
- **F1-score** : it measures the harmonical mean of recall and precision. It combines these two measures and is calculated as $\frac{2 \times TP}{2 \times TP + FP + FN}$.

In the case of multi-class classification, the principle is the same. We need to find the True Positive, True Negative, False Positive, and False Negative samples for each class. From those, we are able to calculate the relevant metrics for each individual class based on the equation written above. A simple example involving 3 classes is represented in Figure 2.12.

		True Class		
		Apple	Orange	Mango
Predicted Class	Apple	7	8	9
	Orange	1	2	3
	Mango	3	2	1

• $TP = 7$
• $TN = (2+3+2+1) = 8$
• $FP = (8+9) = 17$
• $FN = (1+3) = 4$

Figure 2.12: Confusion matrix for a multi-class classification task. TP, TN, FP, and FN need to be found for each class to calculate the metrics related to the specific class. This example makes the calculation for the 'Apple' class [41].

2.4 Computational complexity metric

It is possible to measure the computational complexity of a model. This metric is interesting to compare different architectures. A higher computational complexity leads to a larger latency and more energy consumed for the inference. We express the metric in the number of **Multiply-Accumulate operations (MACC)**. The reason is that many operations in a neural network are dot products involving multiplications and additions :

$$y = \mathbf{w}[0] \times \mathbf{x}[0] + w[1] \times x[1] + w[2] \times x[2] + \dots + w[n-1] \times x[n-1] \quad (2.6)$$

where $\mathbf{w}$ is the weight vector of the layer, $\mathbf{x}$ is the input vector of the layer, $\mathbf{n}$ the number of dimensions of the input data, and $\mathbf{y}$ is the layer scalar output.

The term in bold in Equation 2.6 counts for 1 MACC since it involves one multiplication and one addition. The complete equation thus involves $\mathbf{n}$ MACC. We could argue that it is not true since there are only $\mathbf{n}-1$ additions in Equation 2.6 but the number of MACC is an approximation. For the remainder of this section, the bias set of each layer is neglected as it is only an addition operation.

Let us see the number of MACC operations of several typical layers in convolutional neural networks. The explanation is based on [42]. In the remainder of this section : $\mathbf{H}$, $\mathbf{W}$, $\mathbf{C}$, $\mathbf{in}$, and $\mathbf{out}$ mean respectively the height, width, number of channels, input, output. Therefore, for instance, H_{in} means the height of the input feature map.

- **Convolutional layer** : it takes feature maps of shape (H_{in}, W_{in}, C_{in}) as input and outputs feature maps of size $(H_{out}, W_{out}, C_{out})$. For that, it uses filters of size (k, k) considering a square kernel. The number of MACC operations is given by $\mathbf{k} \times \mathbf{k} \times \mathbf{C}_{in} \times \mathbf{H}_{out} \times \mathbf{W}_{out} \times \mathbf{C}_{out}$. Indeed, for each pixel of each output feature map $(H_{out} \times W_{out} \times C_{out})$, we compute a dot product (one MACC) between the kernel weights and the input's receptive field $(k \times k)$. We perform this operation for each input feature map (C_{in}) ;
- **Depthwise separable convolution layer** : the layer is based on two operations ; the depthwise convolution and the pointwise convolution. The first one is a convolution applied on each input channel separately. The number of filters is equal to the number of input feature maps and each filter has its corresponding input channel. It thus yields a number of output feature maps equal to the number of input feature maps. Therefore, its number of MACC operations is $k \times k \times C_{in} \times H_{out} \times W_{out}$. The pointwise layer is a convolution with a kernel of size $(1,1)$ on the output of the depthwise layer. It thus yields $C_{in} \times C_{out} \times H_{out} \times W_{out}$ MACC operations. The complete operations give a number of MACC equal to $(\mathbf{C}_{in} \times \mathbf{H}_{out} \times \mathbf{W}_{out}) \times (\mathbf{k} \times \mathbf{k} + \mathbf{C}_{out})$. Dividing this result by the one from the standard convolution allows to obtain a factor $\frac{1}{k^2} + \frac{1}{C_{out}}$. Therefore, for instance, a depthwise separable convolution with a kernel size 3×3 and 100 output feature maps involves $0.12 \times$ the number of MACC operations of the standard convolution. This is a significant reduction of MACC operations between the two layers ;
- **Fully connected layer** : it takes a vector of length $\mathbf{I}$ as input and outputs a vector of length $\mathbf{O}$. Because each input is connected to all output neurons, the weight matrix of the layer contains $\mathbf{I} \times \mathbf{O}$ values. The layer purpose is to compute a matrix multiplication between the weights and the inputs. In fact, matrix multiplication is simply many dot products. Each of them happens between the input of length $\mathbf{I}$ and one column of the weight matrix of length $\mathbf{I}$ as well. These $\mathbf{I}$ dot products must be done for the $\mathbf{O}$ columns of the weight matrix. The number of MACC operations is therefore given by $\mathbf{I} \times \mathbf{O}$.

Other layers exist in a network but the most important and computationally expensive ones have been proposed. Note that the activation layers are not measured in MACC as they do not use dot products. However, their impact is so small compared to the proposed operations that it is simply neglected.

In practice, specific Python packages such as *keras-flops* [43] allow doing the calculation easily based on the proposed equations.

2.5 Overfitting in neural networks and prevention

Overfitting is the phenomenon in *Deep Learning* for which the trained model can give high performance on the training dataset but generalizes poorly on unseen data. In other words, the model memorizes the training data too well in such a way it is unable to generalize well on the validation and test sets. For instance, the model may have learned specific images from the training set instead of features from the images. Overfitting can be detected by analyzing the accuracy and loss curves from the training and validation sets, as depicted in Figure 2.13.

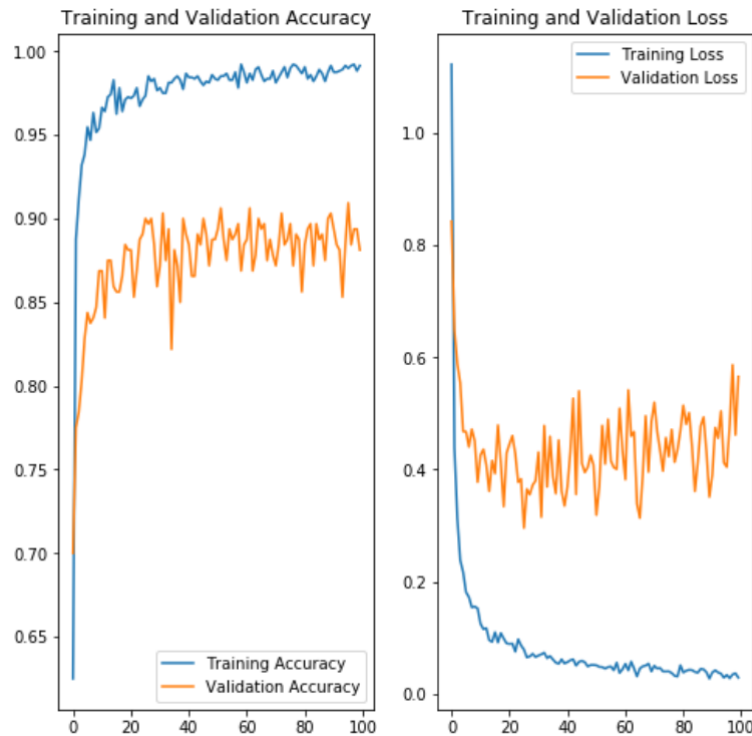


Figure 2.13: Overfitting phenomenon in a classification task. The training accuracy is 11% larger than the validation accuracy and the training loss is $19.5\times$ smaller than the validation loss. These two behaviors involve the overfitting of the model [44].

Overfitting can be mitigated with a lot of techniques among two categories : data augmentation and regularization. Let us see each technique in detail :

2.5.1 Data augmentation technique

Data augmentation is the process of artificially improving the training dataset by creating different versions of the original training samples. It is a technique used a lot nowadays since it allows to

increase the size of a dataset without the need to obtain more data from the real world, as it is not always possible. It allows to reduce overfitting and improves the generalization of the model by presenting to the network different versions of the training samples, thus increasing the diversity of the training dataset [45].

2.5.1.1 Image-based techniques

There exists a lot of transformations applied to image data. The most common ones are presented below. Note that the random transformations are applied with range parameters.

- Random translations in the x and y axes ;
- Random zoom ;
- Random rotation ;
- Random shearing in the x and y axes ;
- Random horizontal and/or vertical flips ;
- Random brightening ;
- Random noise injection ;
- Random cropping.

2.5.1.2 Audio-based techniques

Audio data augmentation techniques also exist in the literature. They can be performed on the raw audio sample or the pre-processed spectral feature. For implementation facilities, we choose the first proposition. However, several augmentation techniques are only possible on the spectral feature, as it will be seen. As for the image augmentation techniques, a lot of them are possible and we focus our analysis on the most known and efficient ones from the literature [46]. As for image augmentation techniques, the random transformations depend on range parameters.

- **Time Shift** : it shifts the data randomly towards the right or the left. What is lost on one side of the recording is recovered on the other side. There exists a difference in the literature for which several people tend to say that what is lost in one side is not recovered in the other side but zero padded. However, we choose the first idea, which is represented in Figure 2.14 ;

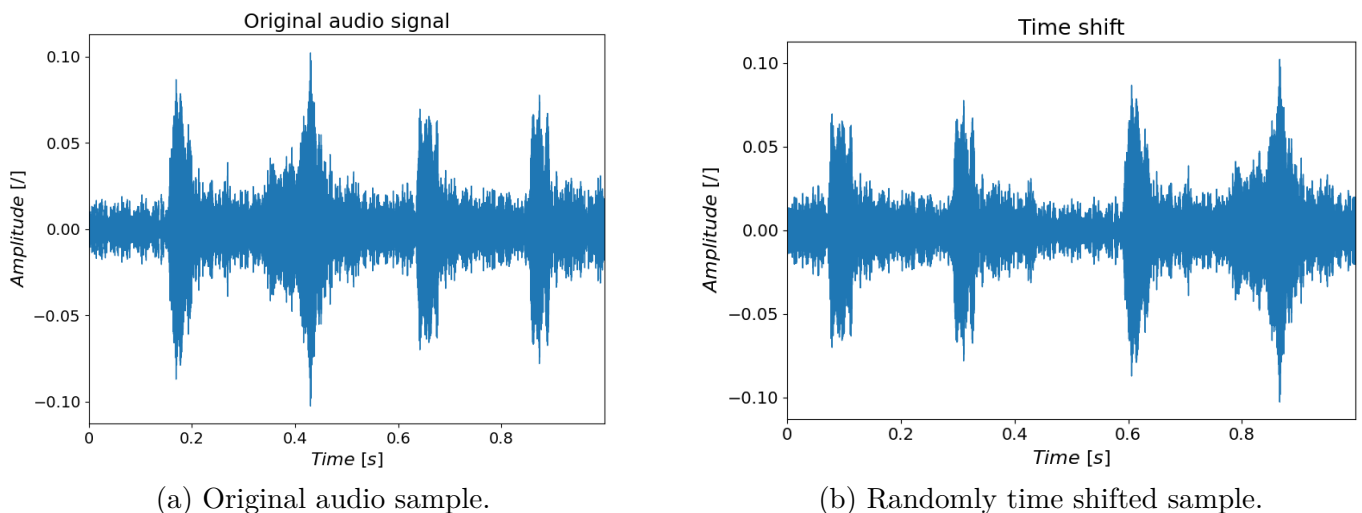


Figure 2.14: Time shift audio augmentation technique.

- **Noise Injection** : it superposes noise on the audio sample. Usually, random Gaussian noise with a given standard deviation is chosen. The Noise injection technique is shown in Figure 2.15 ;

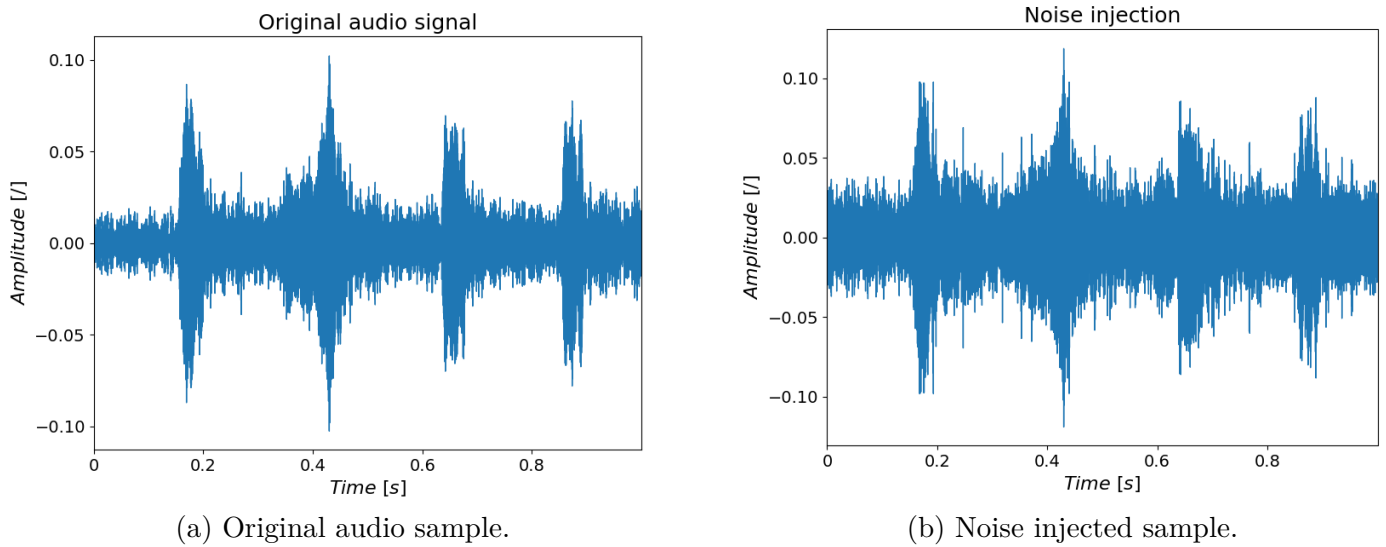


Figure 2.15: Noise injection audio augmentation technique.

- **Pitch Shift** : it changes the pitch of the sound by a given factor. It is shown in Figure 2.16 where the pitch is shifted up by two semitones ;

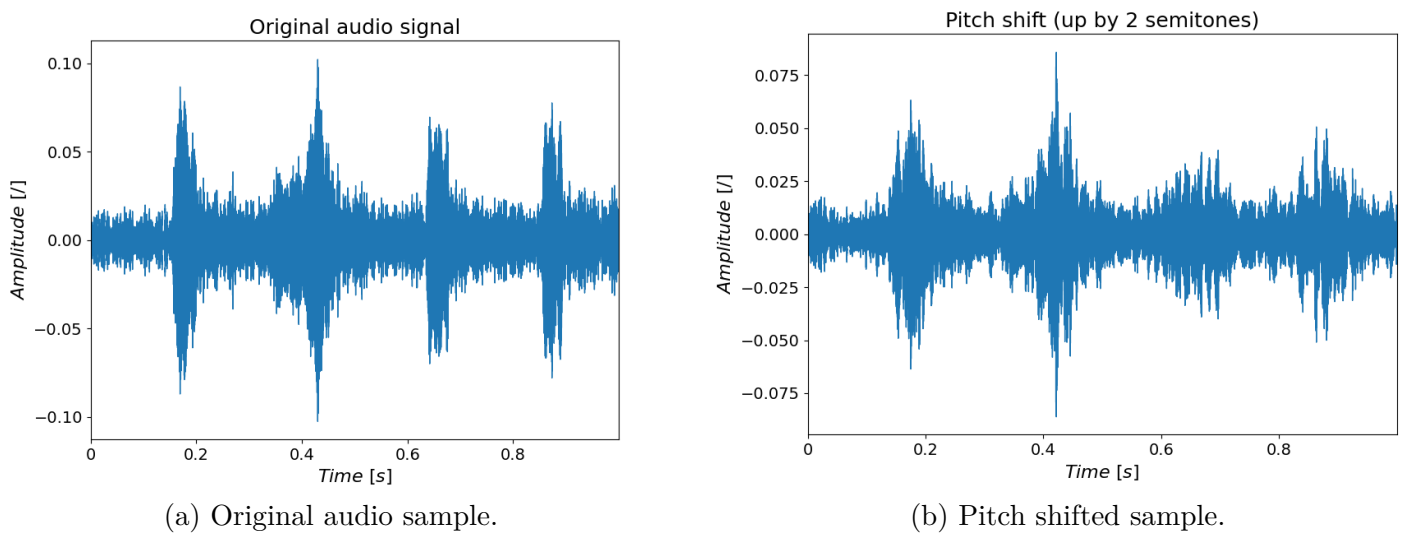


Figure 2.16: Pitch shift audio augmentation technique.

- **Time Stretch** : it modifies the speed of the signal by a given factor. If the signal speed is increased, the end of the sample is zero padded to keep the original audio length. The technique is shown in Figure 2.17 with a speed increase of 15% ;

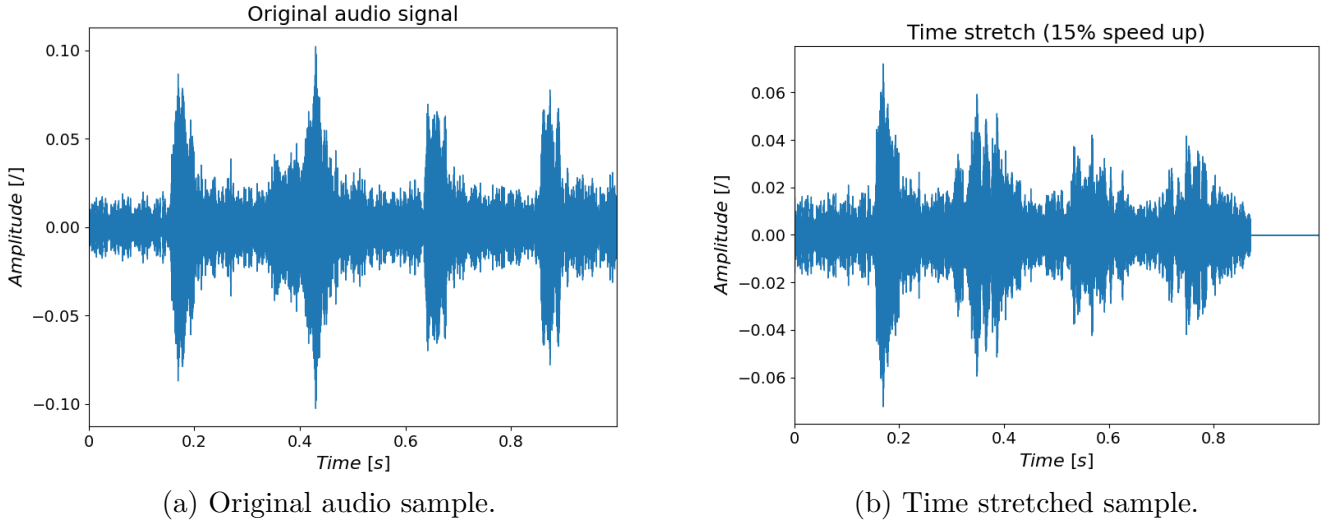


Figure 2.17: Time stretch audio augmentation technique.

- **Mix-Up** : this is a recent audio augmentation technique that yields impressive performance in a lot of audio classification tasks [47, 48]. Its principle is fairly simple as it creates a new sample based on two others of different classes. The new sample, as well as the new label, are obtained by the following equations :

$$x = \alpha \times x_i + (1 - \alpha) \times x_j \quad (2.7)$$

$$y = \alpha \times y_i + (1 - \alpha) \times y_j \quad (2.8)$$

where (x_i, y_i) and (x_j, y_j) are two samples that are randomly selected from the training dataset, α is a parameter lying in $[0 ; 1]$ and that is drawn from a Beta probability density function $\beta(\lambda, \lambda)$ where $\lambda \in [0, +\infty]$ [48]. Note that the labels are one-hot encoded and the new sample has the class which has the biggest probability in the new label. An example of a combination of a bird and animal sample is shown in Figure 2.18. The α coefficient value was equal to 0.57, leading to a new bird sample.

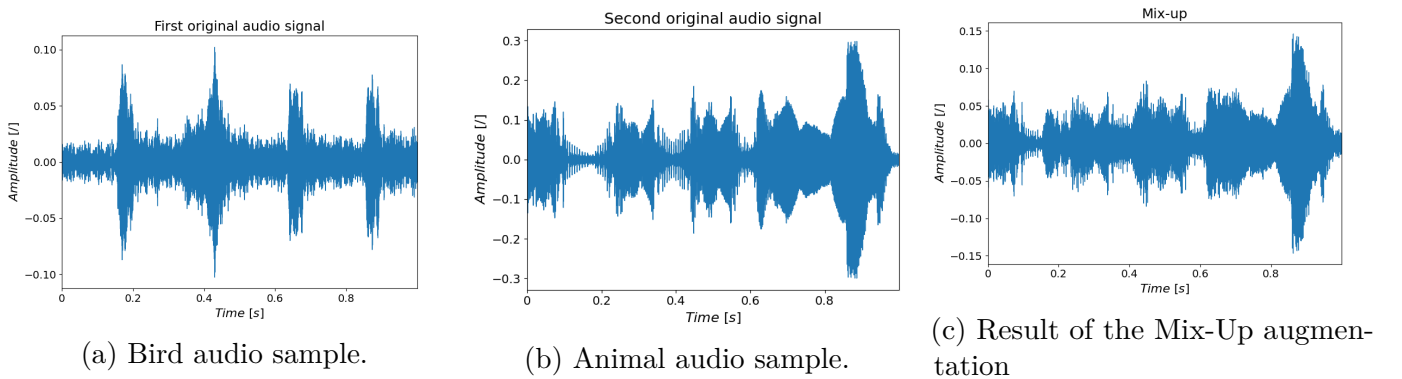
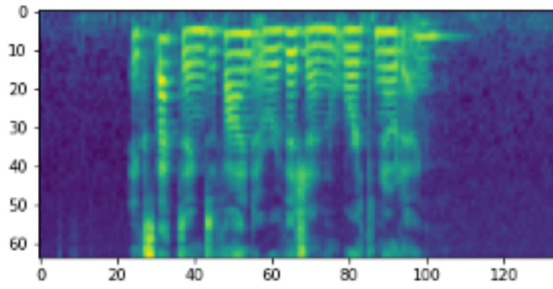
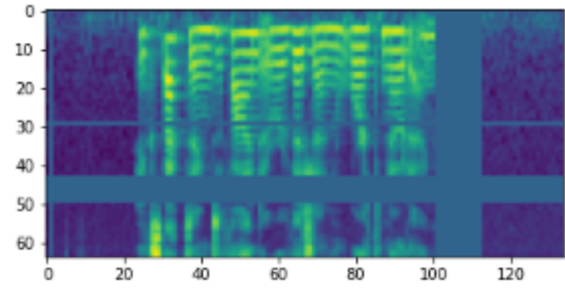


Figure 2.18: Mix-Up audio augmentation technique

- **Frequency & Time Masks** : this is a recent technique that can only be performed on the spectral feature [49]. It consists in hiding some portions of the spectral feature in one or both directions (time and frequency). This is represented in Figure 2.19 ;



(a) Original audio spectrogram.



(b) Frequency & Time masked spectrogram.

Figure 2.19: Frequency & Time Masks audio augmentation technique [46].

2.5.2 Regularization techniques

According to [50], Ian Goodfellow says that "regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error". Let us see some known techniques.

2.5.2.1 Complexity of the network

Complex models tend to overfit because they contain many parameters, making them able to learn a lot of irrelevant features from the input. A simple way to reduce overfitting is thus to limit the model capacity such that it does not memorize irrelevant patterns present in the training data.

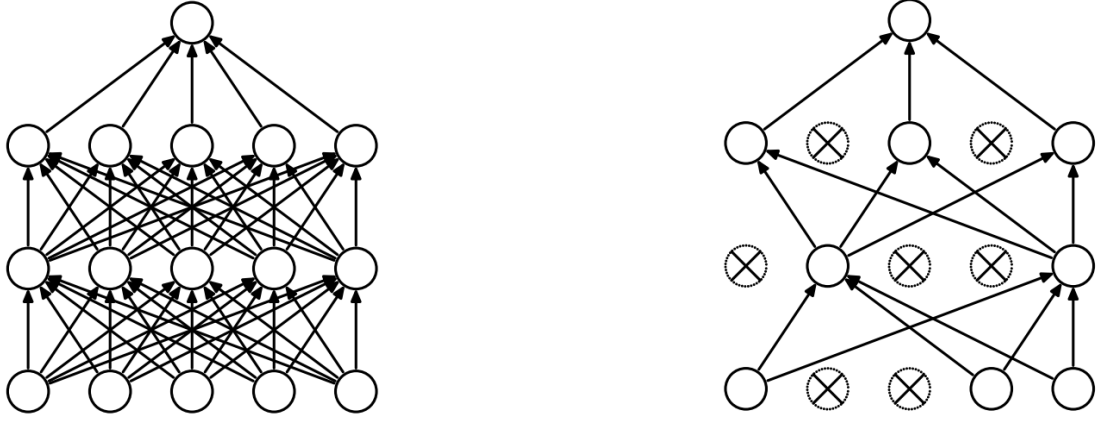
2.5.2.2 Weights regularization

It allows to constraint the weights of the neural network. Basically, the loss function is modified to constraint the weight values. Two of the most known weights regularization are presented, based on [50] [51] :

- The L2 regularization : "it adds a L2-norm term in the loss function allowing to penalize big squares values of the parameters".
- The L1 regularization : "it adds a L1-norm term in the loss function allowing to penalize big absolute values of the parameters".

2.5.2.3 Dropout

This is a technique that allows dropping several activations of neurons. The dropout rate controls the number of units to set to 0 in a specific layer. This procedure is only applied during training and "prevents units from co-adapting too much" [52]. This technique yields state-of-the-art performance over other regularization methods. For instance, on the MNIST dataset, the state-of-the-art performance at that time was an error of 1.6%. Using dropout allowed to reduce the error to 1.35%. The dropout technique improves several other benchmark datasets such as ImageNet and CIFAR-100 [52]. The principle of a dropout layer is shown in Figure 2.20.



(a) Standard neural network with two hidden layers. (b) Standard neural network after applying dropout.

Figure 2.20: Dropout principle. On the left is represented a simple neural network with 2 hidden layers. On the right is shown the same network after applying dropout. Several connections, controlled by the dropout rate of the layer, are removed to create a thinner network at training time [52].

2.5.2.4 Batch normalization

It is a process that adds layers in a deep neural network and "performs the standardizing and normalizing operations on the input of a layer coming from a previous layer" [53]. It basically allows normalizing the data inside the network. It is known to train neural networks faster and to stabilize them during the training process [54]. The same paper demonstrates that an Inception architecture using batch normalization before each activation layer is able to obtain a boost of 0.5% in accuracy on the *ImageNet* dataset while requiring $2.33\times$ less training time compared to the baseline Inception network. Moreover, they show that simply using batch normalization to a network does not bring full benefits to the method. They thus modify further the architecture and the training by removing dropout layers and increasing the learning rate for instance. It yields a boost in classification accuracy of 2.6% on *ImageNet* while taking $11.48\times$ less time to train compared to the initial Inception model [54].

The batch normalization layer works in several steps. The in-depth explanation is based on [54, 55]. First, it takes a batch of inputs and calculates the mean of the batch for the hidden activations following this equation :

$$\mu = \frac{\sum_{i=1}^n a_i}{n} \quad (2.9)$$

where μ is a vector representing the mean of the batch of inputs for each hidden activation, a_i is a vector representing all hidden activations instanciated with the input i of the batch and n is the number of inputs in the batch.

Secondly, it calculates the standard deviation of the mini-batch for the hidden activations :

$$\sigma = \sqrt{\frac{\sum_{i=1}^n (a_i - \mu)^2}{n}} \quad (2.10)$$

where σ is a vector representing the standard deviation of the batch of inputs for each hidden activation.

From now on, we have at our disposal the mean and the standard deviation of the hidden activations based on the mini-batch. We can thus normalize these hidden activations of the layer by setting their

mean to 0 and their standard deviation to 1. This is done following Equation 2.11.

$$a_{i_{normalized}} = \frac{a_i - \mu}{\sigma + \epsilon} \quad (2.11)$$

where $a_{i_{normalized}}$ is a vector representing the i_{th} normalized activation of the layer and ϵ is a small floating-point value known as the smoothing factor that ensures numerical stability by avoiding dividing by zero.

Finally, the normalized hidden activations are scaled and shifted following Equation 2.12.

$$y_i = \gamma \times a_{i_{normalized}} + \beta \quad (2.12)$$

where γ is the scaling factor and β is the shifting factor. These two parameters vectors (they have the same number of elements as the number of hidden activations in the layer) are learned in a batch normalization layer to optimize the normalization of each mini-batch.

2.5.2.5 Others

Weight initialization is sometimes seen as a way to prevent overfitting. The idea is to select a prior distribution for the weights of the network to try to reach more closely the global minimum of the cost function. Early stopping regularization is also a technique to prevent overfitting and is explained in Chapter 5.

2.6 Audio Classification

Audio classification is the task of classifying audio samples among a given set of classes. Many tasks in this field exist and a deep explanation is proposed in Chapter 3. In this section, we focus on the definition of an audio sample as well as its primary properties. Then, we discuss the typical signal processing chain to classify an audio sample. Finally, we analyze in detail the spectral features of an audio waveform typically given in input to the CNN.

2.6.1 Analog and digital sound

The sound is a variation of pressure over a period of time induced by a vibrating object. Sound is an energy that is transported typically over the air. The amount of work to generate this energy sets the **amplitude** of a sound. The **frequency** of a sound is the total number of waves produced in a duration of one second [56]. The sound is an analog (continuous values) waveform that is recorded through a microphone. However, computers and deep learning models work with digitized numbers. Therefore, the sound is processed through an Analog-To-Digital Converter (ADC) to digitize it (discrete values). The process is represented in Figure 2.21.

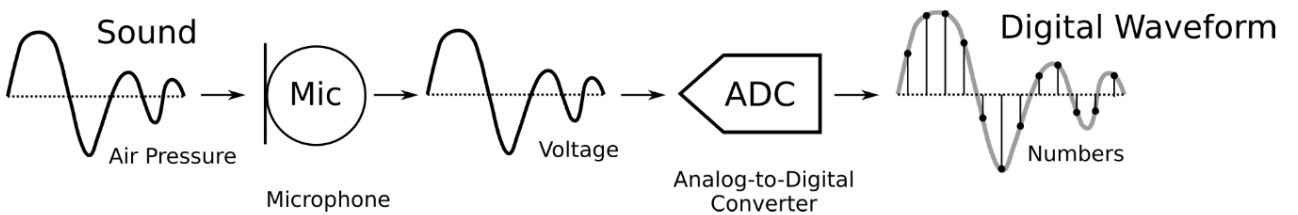


Figure 2.21: Capture of an analog sound with a microphone, later converted into a digital representation with the ADC block [30].

The digitization process involves a quantization in the time and the frequency domain of the analog wave. On the one hand, the resolution or bit-depth is the number of bits that are used for the conversion in the amplitude domain. Typically, a continuous number takes a value among two at the power of the bit-depth minus one possible values describing the full scale of the analog waveform. The resolution is important as it is responsible for an error due to the conversion process, known as the quantization error. Larger is the resolution, smaller is this error, and bigger is the quality of the sound. As an example, imagine a continuous value equal to 1.3256 V, a resolution of 4 bits and the full scale of the analog wave equal to 3.3 V. In that case, each digitized value has at least a difference of $\frac{3.3}{2^4-1} = 0.22$ called the resolution. Therefore, the continuous value is going to be rounded to 1.32 (0.22×6), leading to a small error in that case. If the continuous value were 1.4256 V, it would also be rounded to 1.32 because it is the closest digital value, leading to a much higher error in that case. On the other hand, the sample rate is a quantization in the frequency domain. It is the number of samples that are taken each second from the analog waveform. A famous theorem, known as the Nyquist-Shannon theorem, states that the sampling rate defines the Nyquist frequency with :

$$f_s = \frac{SR}{2} \quad (2.13)$$

where f_s is the Nyquist frequency and SR is the sample rate, also called the sampling frequency. The Nyquist frequency is the maximal frequency that can be analyzed in the signal. A typical value of resolution is 16 bits, while a typical sampling frequency is 44100 Hz, as used in Audio CD format. With these specific parameters, the major part of the acoustic information is kept as humans do not perceive sounds higher than 22 kHz in practice. The human frequency range is typically in the range [20 - 20000] Hz [57].

The sound can be recorded in mono or in stereo. A mono recording means that the data contains only 1 dimension because the sound was recorded by only one device. However, in a stereo recording, two microphones are used to collect the sound. Therefore, the digitized data has two channels (dimensions). Typically, in deep learning, we prefer to use only one channel data.

Sound can be stored in an uncompressed format such as the WAV PCM. It thus means the digitized sound after the quantization process has not been modified. However, usually, the sound is compressed to be stored more efficiently since it reduces the size of the data. The compression algorithms can be divided into two categories :

- Lossless compression : the most known format is probably the FLAC codec. The idea is to store more efficiently the redundancy in the data and to store the differences between prediction and actual samples. No information in the temporal or frequency domain is lost with this category ;
- Lossy compression : this is the most used type of compression nowadays. It allows reducing the size of the data much more than with a lossless compression. However, it comes with a price to pay : a loss of audio information. Typically, this loss of information is not perceptible for most people if we compare it to the real uncompressed sound. However, for deep learning purposes, this loss of information can be serious for the training of the network. One audio classification task which is seriously impacted by lossy codecs is Bird Classification because the birds do not hear as humans do. It is therefore avoided as much as possible in the machine learning field. Examples of lossy compression codecs are MP3, AAC or OGG. Figure 2.22 shows a loss of high-frequency content between an uncompressed square wave sound and its lossy compression with the MP3 codec.

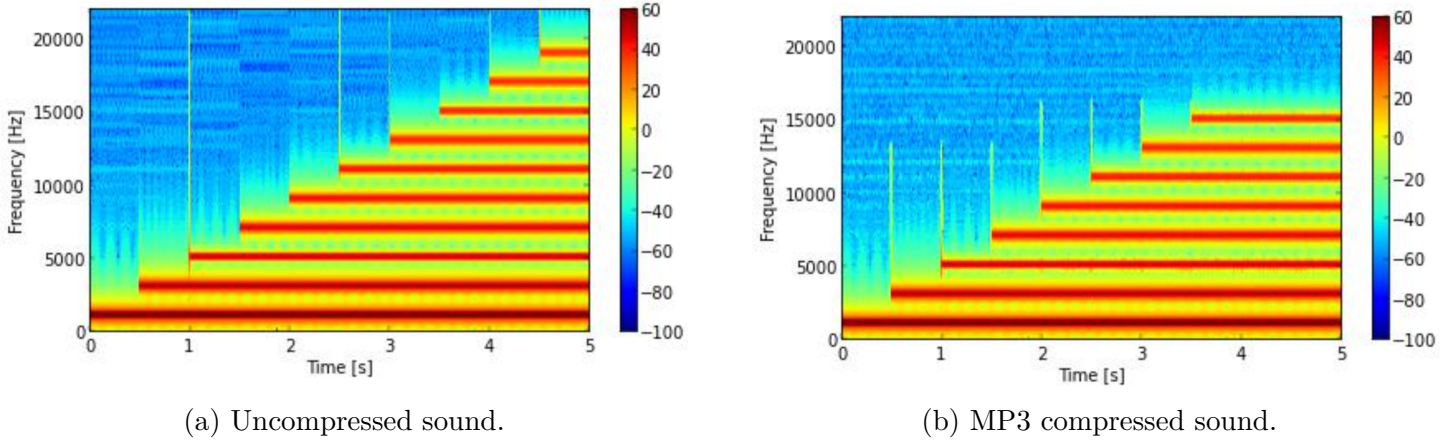


Figure 2.22: Loss of information in the high frequencies due to the MP3 codec. The sound is a square wave with harmonics 1, 3, 5, ..., 19 [58].

2.6.2 Audio classification chain

The audio classification pipeline is usually performed in the same manner regardless of the task. This is true both for the training of the neural network and the prediction of the latter. It is represented in Figure 2.23 for a keyword spotting audio classification task. First, we gather an audio sample, for which the amplitude changes as a function of time. Then, we extract a spectral feature out of this temporal signal. This feature is an image for which one dimension is the time and the other is the frequency. Spectral features are thus a time-frequency representation of the initial audio waveform. In step 3, the extracted image is given to the convolutional neural network, either for the training or the prediction that happens in step 4.

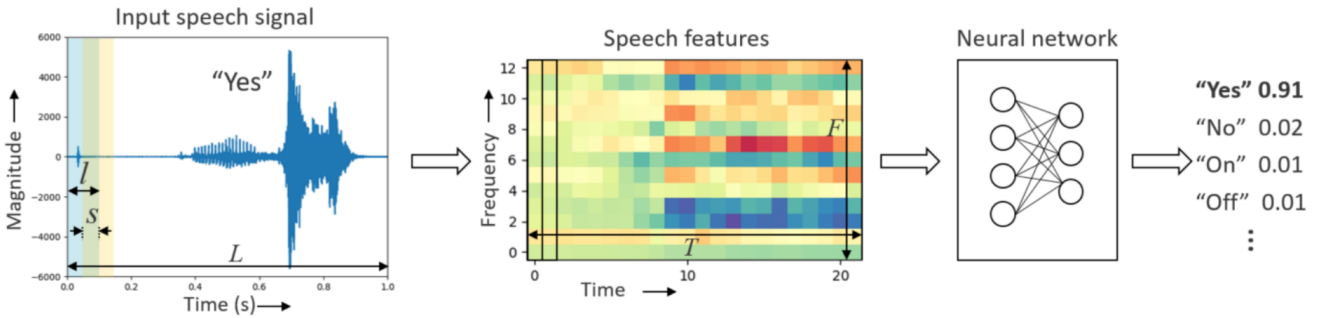


Figure 2.23: Typical chain for classifying an audio sample [6].

There exists many spectral features that are analyzed in the next subsection. Of course, multiple spectral features can be extracted from the audio sample and used together for the classification. Note that the used spectral features are different with respect to the classification task. Moreover, it is possible not to compute these spectral features and directly use the raw audio waveform. It requires, however, another architecture than the CNN introduced in Section 2.1. The first convolutional layer must be a 1D convolutional layer. It is also possible to use other architectures accepting 1D signals as input. Finally, the pipeline presented in Figure 2.23 can be replaced or added by the computation of acoustic features, which are one-dimensional vector. From these specific features, a 1D CNN or other architectures can be used. All these differences are fully detailed in Chapter 3, addressing the state-of-the-art audio classification.

2.6.3 Audio Spectral Features

In this section, we present seven different features that can be used as input to a deep learning model. Each feature, except the *scalogram*, is always derived from the first one, referred to as the *spectrogram*. We therefore briefly discuss the pipeline to process an audio sample into a spectrogram. For the other features, we add relevant information showing the difference with the spectrogram feature. To visually represent the spectral features, we use a gunshot audio waveform represented in the following figure.

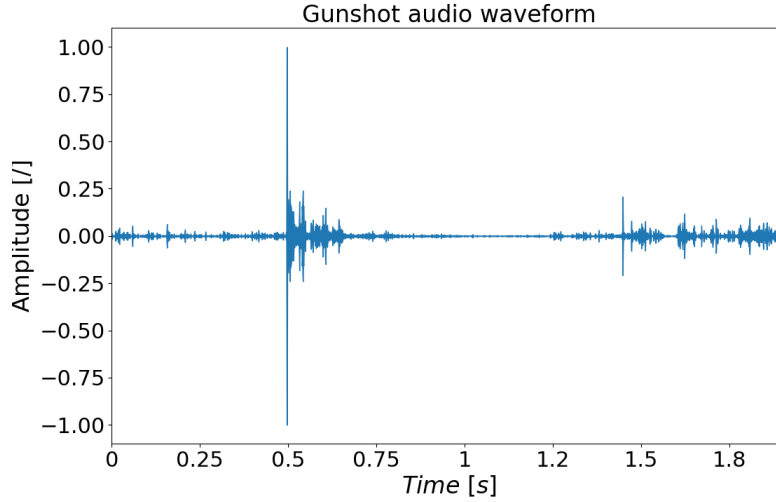


Figure 2.24: Gunshot 2-seconds audio sample.

2.6.3.1 Spectrogram

The spectrogram is a spectral feature represented as an image. It is a time-frequency representation of the audio waveform, with the time on the x-axis and the frequency on the y-axis. Concretely, a spectrogram represents the power content of the signal (amplitude squared) at a specific frequency band and temporal frame. A spectrogram is obtained by performing a **Short-Time Fourier Transform** [59, 60]. The process involves cutting the audio signal into multiple chunks or frames using a windowing function. Then, a discrete Fourier transform is performed on each frame to obtain the frequency spectrum. Those different spectra are then concatenated to form a 2D image. Due to the inner working of the STFT, a spectrogram have many parameters to optimize : the windowing function, the Fast Fourier Transform (FFT) size, and the frames overlap proportion are such examples. These parameters affect the size of the spectrogram. For instance, the number of temporal frames is given by :

$$Temporal_{frames} = \frac{SR - FS}{Overlapping} + 1 \quad (2.14)$$

where SR is the sample rate, FS the frame size (often equal to the FFT size), and Overlapping the number of overlapping frames.

Figure 2.25 represents the spectrogram of the gunshot audio. The colorbar represents the magnitude of a frequency band at a specific temporal frame.

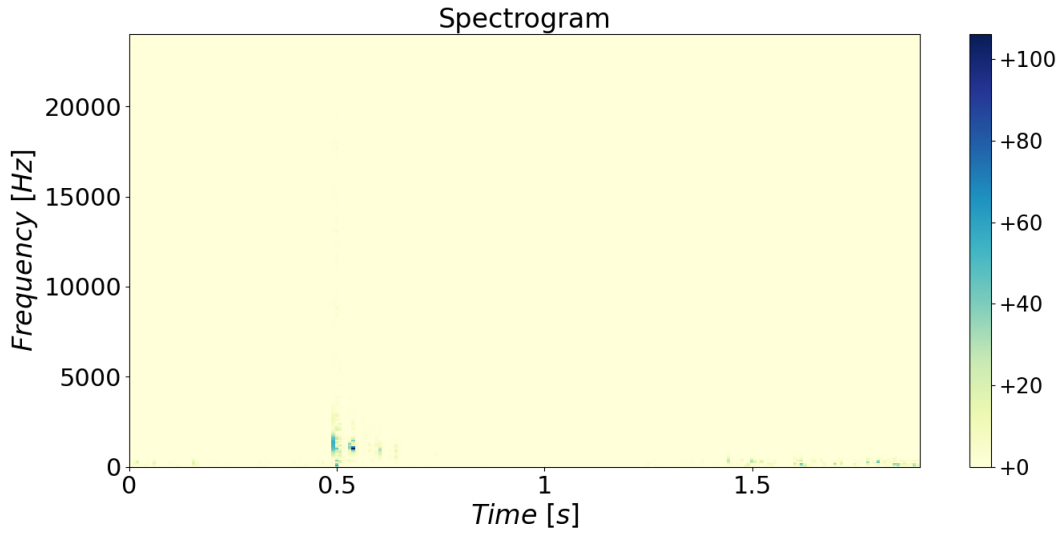


Figure 2.25: Gunshot spectrogram.

2.6.3.2 Log-amplitude spectrogram

The log-amplitude spectrogram performs one more computation compared to the original spectrogram. We convert the power spectrogram to decibel units to mimic the human auditory system. The log-amplitude spectrogram is represented in Figure 2.26.

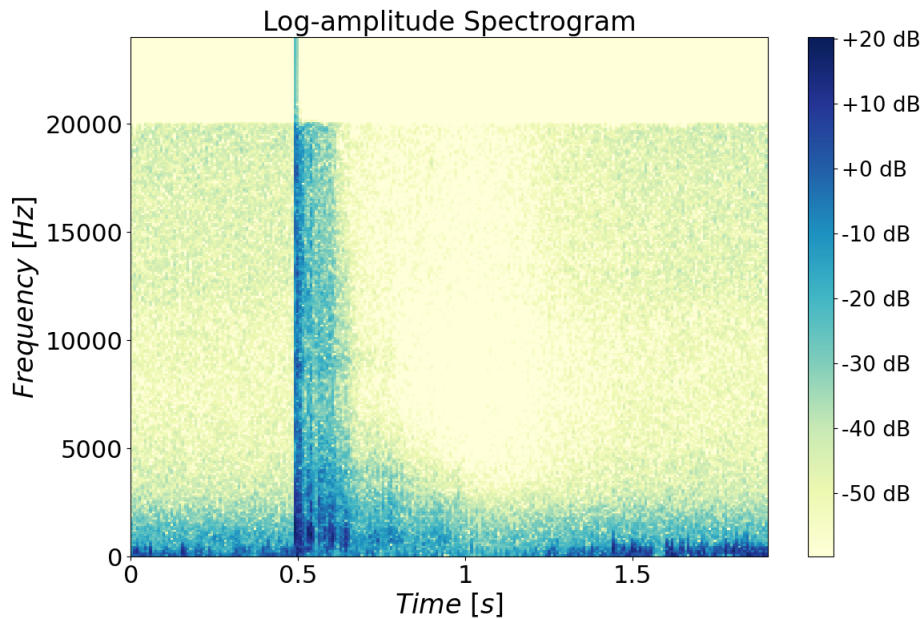


Figure 2.26: Gunshot log-amplitude spectrogram.

2.6.3.3 Mel spectrogram

The mel spectrogram introduces the **mel scale** represented in Figure 2.27. This is a logarithmic scale allowing to map frequencies from Hz units to frequency in mels units. Concretely, the mel scale allows getting a perceptually relevant frequency representation that we do not have in spectrograms. This feature is due to the fact that humans do not perceive the frequency linearly. For instance, the human ear does not perceive the same sound between $[60 ; 260]$ Hz and $[1600 ; 1800]$ Hz. The mel spectrogram is the power spectrum for which the frequencies have been converted in mels. This feature is represented in Figure 2.28.

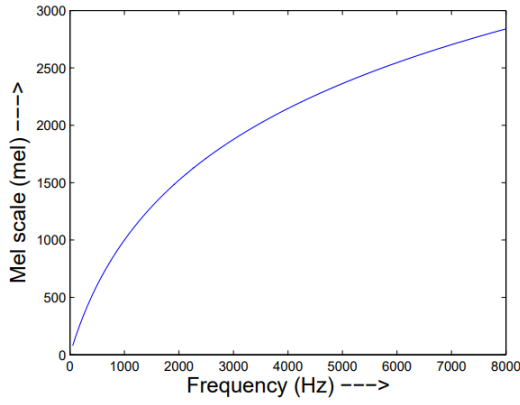


Figure 2.27: Mel scale [61]

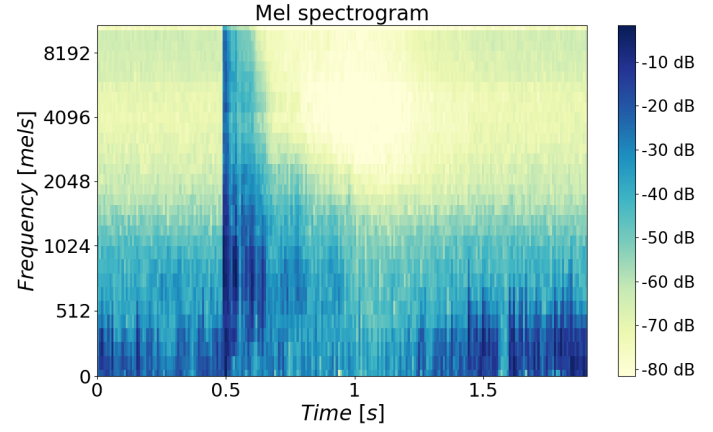


Figure 2.28: Gunshot mel spectrogram.

2.6.3.4 MFCC

The MFCC feature stands for Mel-frequency cepstral coefficients. They are computed by applying a discrete cosine transform to the logarithm of the mel spectrogram. The MFCC feature set is a lossy format containing only several coefficients representing the signal. The feature is shown in the following figure.

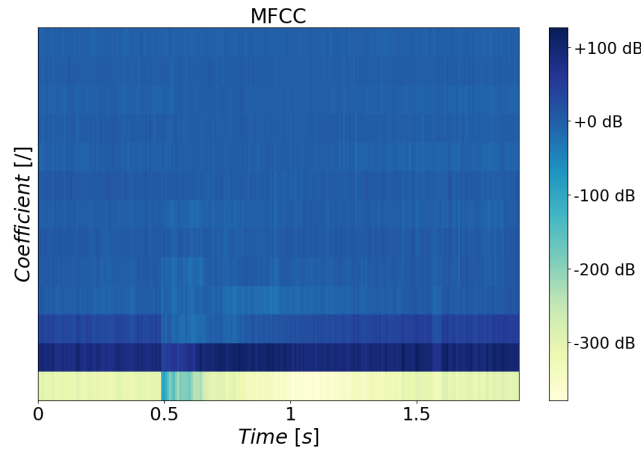


Figure 2.29: Gunshot MFCC.

2.6.3.5 Other less known features

Cochleagram The cochleagram is close to the mel spectrogram except that it does not use the same filter bank. Concretely, to compute a mel spectrogram, mel-filter banks must be applied to the original spectrogram. In this case, we apply gammatone filter banks to the spectrogram. The cochleagram, also referred to as the gammatone spectrogram, tends to mimic the cochlea of the human auditory system.

GFCC GFCC stands for Gammatone-frequency cepstral coefficients. They are computed by performing a discrete cosine transform on the cochleagram. These are similar to the MFCC feature.

Scalogram A scalogram is completely different from a spectrogram. Without entering into the details, the scalogram is performed by computing a continuous wavelet transform and this feature represents the energy of each coefficient of the wavelet transform.

This chapter studies the available audio classification tasks as well as their state-of-the-art implementations of convolutional neural networks and variants. It aims to be aware of state-of-the-art models and features used in different audio studies. We are also interested in the type of model (image-based, audio-based, energy-efficient) used in the state-of-the-art for each task. To that purpose, we divide this chapter into four categories representing the most known audio classification tasks. For each of them, we first present the famous benchmark datasets. Then, we analyze the state-of-the-art works related to several of these benchmarks.

3.1 Environmental Sound Classification

It is an active area in audio classification and has progressed a lot in a few years. The task is to classify environmental sounds and noises.

3.1.1 Datasets

The *ESC-50* dataset [62] is a labeled collection of 2000 5-seconds-long environmental audio recordings organized into 50 categories represented in Figure 3.1. Each class contains precisely 40 samples. The recordings have one audio channel and a sample rate equal to 44.1 kHz. They are all compressed with an Ogg Vorbis compression at 192 kbit/s. The dataset was created using sounds from the *Freesound* project [63]. A reduced dataset, referred to as *ESC-10*, containing only ten categories among the fifty, is also known in the literature. As mentioned in [62], around 4000 people have manually tested both datasets to obtain human classification accuracy. It was estimated to be 81.30% on *ESC-50* and 95.7% on *ESC-10*. A lot of works have tried to get the highest accuracy on the *ESC-50* benchmark. Many of them are available in [64].

Animals	Natural soundscapes & water sounds	Human, non-speech sounds	Interior/domestic sounds	Exterior/urban noises
<u>Dog</u>	<u>Rain</u>	<u>Crying baby</u>	Door knock	<u>Helicopter</u>
<u>Rooster</u>	<u>Sea waves</u>	<u>Sneezing</u>	Mouse click	<u>Chainsaw</u>
Pig	<u>Crackling fire</u>	Clapping	Keyboard typing	Siren
Cow	Crickets	Breathing	Door, wood creaks	Car horn
Frog	Chirping birds	Coughing	Can opening	Engine
Cat	Water drops	Footsteps	Washing machine	Train
Hen	Wind	Laughing	Vacuum cleaner	Church bells
Insects (flying)	Pouring water	Brushing teeth	Clock alarm	Airplane
Sheep	Toilet flush	Snoring	<u>Clock tick</u>	Fireworks
Crow	Thunderstorm	Drinking, sipping	Glass breaking	Hand saw

Figure 3.1: The 50 categories of the *ESC-50* dataset [64]. The categories underlined in red are part of the *ESC-10* dataset.

The *UrbanSound* dataset [65] is a collection of 1302 audio recordings of varying length and audio properties. All samples were obtained and labeled from the *Freesound* project. It consists of 10 categories selected due to their frequent apparition in urban noise complaints. For research and benchmark purposes, a subset of this dataset referred to as the *UrbanSound8K* dataset has been created. Based on [66], which showed that 4-seconds sounds were sufficient for people to identify environmental noises with an accuracy of 82%, the *UrbanSound8K* dataset is composed of 8732 4-seconds recordings. We show the categories and the number of samples per class in Figure 3.2.

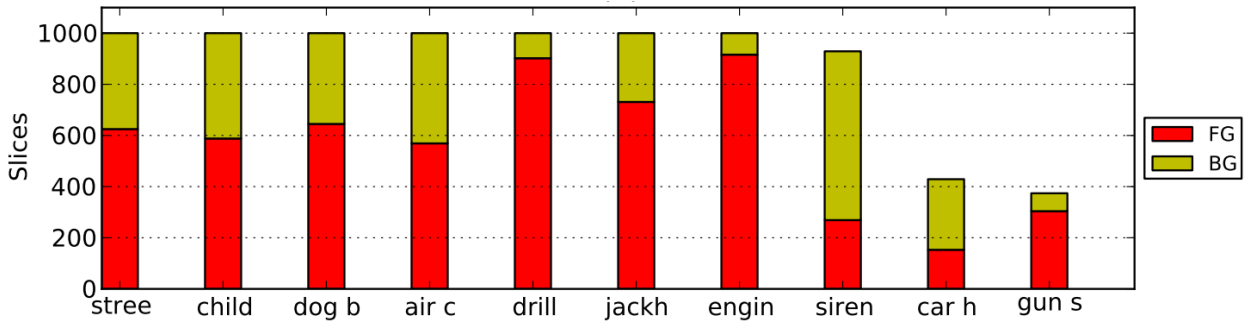


Figure 3.2: The ten categories of the *UrbanSound8K* dataset are: air conditioner, car horn, children playing, dog bark, drilling, engine idling, gunshot, jackhammer, siren, and street music. The number of samples per category is represented in the figure [65].

The *SONYC-UST-V2* dataset [67] aims for real-world urban noise monitoring and is made of 18510 10-seconds audio recordings. The sounds have been collected through the SONYC acoustic sensor network, an ensemble of more than 50 sensors in the periphery of New York fighting against noise pollution in urban areas. The dataset proposes 23 categories representing frequent causes of noise complaints in New York City. In addition to the recordings, the dataset includes Spatio-temporal information such as the New York block identifier or the accurate recording time. The

SONYC-UST dataset is the first version created for the same objective and consists of 3068 audio recordings from the SONYC acoustic sensor network. The classes are the same as for *SONYC-UST-V2*.

Acoustic Scene Detection [68], general-purpose tagging of sounds [69], and detection of vehicle sounds [70] are related fields to Environmental Sound Classification. Some datasets were released and used for several challenges linked to these tasks.

3.1.2 Related works

As mentioned in [71], the *log-mel spectrogram* spectral feature performed really well on Environmental Sound Classification tasks. It was considered a state-of-the-art feature. We could find the *log spectrogram* and the *CQT* features among the other good performers, while the *MFCC* feature was well below its concurrents. Table 3.1 shows ten of the best state-of-the-art works using convolutional neural networks on the *ESC-10*, *ESC-50*, and *UrbanSound8K* benchmark datasets. The works are classified in decreasing order according to the *ESC-50* benchmark. This table reveals that the best feature types used in Environmental Sound Classification have not changed. The *log-mel spectrogram* feature is present at the top and the bottom of the state-of-the-art, demonstrating its efficiency. Moreover, we can find the *log spectrogram* and the *CQT* features in the best works, yielding one of the highest accuracies. However, it should be noted that some works try to include new features in that field such as the *MFCC*, *GFCC*, *chromagram*, and *log-gammatone spectrogram* feature types.

A lot of these works use spectral features as input to the CNN. The current state-of-the-art performance is achieved by [72] with an out blowing accuracy of 97.57% on *ESC-50* and 99.49% on *UrbanSound8K*. To that purpose, they use deep pre-trained CNNs from the *ImageNet* task: *DenseNet-161* [73] for the first benchmark and *ResNet-152* [74] for the second. The first network is composed of 161 layers, while the second contains 152 layers. The authors perform a specific methodology for the training of the DCNNs. First, they freeze the initial layers of the networks and train the models. Secondly, they unfreeze the initial layers and retrain the models using **discriminative learning**. This technique separates the model layers into three distinct groups and uses an optimal learning rate for each group. The idea is that the depth of the network influences the features that are learned by the network. For instance, the first layers contain small details important for many tasks, and a small learning rate is thus used. Moreover, they also perform known augmentation techniques on the audio samples: pitch shift, time stretch, and silence trimming.

The top-3 state-of-the-art works in Environmental Sound Classification [72, 75, 76] use deep convolutional neural networks, which are highly power-consuming to be trained and operated. Fortunately, several works try to solve this issue by building energy-efficient networks. [77] has built *Micro-ACDNet* by decreasing the complexity and the size of the *ACDNet* architecture [77] by 97%. The model takes as input raw audio data and not spectral features, thus using 1D convolution layers. It demonstrates an accuracy of 81.5% on *ESC-50*, being embedded on a resource-constrained microcontroller.

Finally, more works related to Environmental Sound Classification focus on CNNs taking as input the raw audio waveforms. "A learnable gammatone filterbank layer consisting of 1D kernels represented by a parametric form of the bandpass gammatone filters is proposed for acquiring a time-frequency representation of the raw waveform" in [78]. It yields an impressive result, decreasing by 2.45% the classification error on *ESC-50* compared to the well-known *AcNet* architecture [79], also using the raw audio as a feature.

Work	Feature(s)	ESC-10	ESC-50	UrbanSound8K
Z. Mushtaq & al. (2021) [72]	Mel spectrogram	99.04	97.57	99.49
J. Sharma & al. (2019) [75]	MFCC, GFCC, CQT, chromagram	97.25	95.5	98.6
A. Guzhov & al. (2020) [76]	Log-spectrogram	97	91.5	85.42
Z. Mushtaq & al. (2020) [80]	Log-mel spectrogram	94.94	89.28	95.37
H. Park & al. (2020) [78]	Raw audio	-	88.1	85.8
J. Sharma & al. (2020) [81]	MFCC, GFCC, CQT, chromagram	94.75	87.45	97.52
J. J. Huang & al. (2018) [79]	Raw audio	-	85.65	-
Y. Su & al. (2020) [82]	MFCC, log-mel spectrogram, chromagram, spectral contrast, and tonnetz	-	85.6	93.4
Z. Zhang & al. (2018) [83]	Log-mel spectrogram, log-gammatone spectrogram	91.7	83.9	83.7
M. Mohaimenuz-zaman & al. (2021) [77]	Raw audio	-	81.5	-

Table 3.1: State-of-the-art convolutional neural networks for the *ESC-10*, *ESC-50*, and *UrbanSound8K* datasets. We present the authors of the model and the paper in the first column. The feature types used by the model are mentioned in the second column. The three last columns are the accuracies [%] obtained on the benchmarks datasets. The work highlighted in orange has been implemented on a microcontroller.

3.2 Music Classification

Music Sound Classification is a multifaceted task. Many goals are possible, such as recognizing the genre of music, the composer, or differentiating between several instruments or pieces of music.

3.2.1 Datasets

The *GTZAN* dataset [84] contains 1000 30-seconds audio tracks. Ten genres, each represented by 100 recordings, have to be classified in this task. The recordings are sampled at 22050 Hz with a resolution of 16 bits with one audio channel. The dataset was acquired "in 2000-2001 from a variety of sources including personal CDs, radio, microphone recordings, to represent a variety of recording conditions" as stated in [84]. The different genres of music are listed here: blues, classical, country, disco, hip-hop, jazz, metal, pop, reggae, and rock. It is considered to be the most well-known dataset in music genre recognition. It was used the first time in 2002 by [85].

The *Ballroom* dataset is a set of ballroom dance music in 8 classes, namely Jive, Quickstep, Tango, Waltz, Viennese Waltz, Cha Cha Cha, Samba, and Rumba [86]. The entire dataset contains 698 tracks of 30-seconds long. "The audio quality of this data is quite low. It was originally fetched in

real audio format, with a compression factor of almost 22 with respect to the common 44.1 kHz 16 bits mono WAV format" [86].

The *MagnaTagATune* dataset is a set of 25 863 music clips [87]. Each recording lasts 29 seconds and belongs to one of the 5223 songs, 445 albums, and 230 artists. The clips span a large range of musical genres. [88] states that "each audio clip is supplied with a vector of binary annotations of 188 tags. These annotations are obtained by humans playing the two-player online TagATune game. In this game, the two players are either presented with the same or a different audio clip. Subsequently, they are asked to come up with tags for their specific audio clip. Afterward, players view each other's tags and are asked to decide whether they were presented with the same audio clip. Tags are only assigned when more than two players agreed. The annotations include tags like 'singer', 'no singer', 'violin', 'drums', 'classical', 'jazz'. The top 50 most popular tags are typically used for evaluation to ensure enough training data for each tag."

Other less known datasets exist in the literature, such as the *GuitarSet* dataset, which contains 360 guitar 30-seconds-long recordings [89], the *GiantMIDI-Piano* dataset incorporating 10 854 unique piano solo pieces composed by 2 786 composers [90], or the *MusicNet* dataset providing 330 classical music recordings for an entire duration of 2048 minutes [91].

3.2.2 Related works

Table 3.2 shows nine of the best state-of-the-art works using convolutional neural networks on the *GTZAN* and *Ballroom* benchmark datasets. The works are classified in decreasing order according to the *GTZAN* benchmark. Interestingly, Table 3.2 demonstrates that the *mel spectrogram* and *spectrogram* features yielded state-of-the-art performance on the *GTZAN* dataset and still do on the *Ballroom* dataset. An accuracy of 96.7% is reached on the latter by [92] with the *mel spectrogram* feature. They claim that usual CNN structures are not adequate for music audio classification. They thus use a novel CNN architecture, referred to as the *Bottom-up Broadcast Neural Network*. It is an efficient architecture containing only 180 000 parameters. The change comes from the use of the *Broadcast Module* represented in Figure 3.3. Concretely, it combines multiple convolutions and a pooling layer into one block, called an Inception block from the Inception network [93]. The authors stack these blocks on top of each other, and they link these blocks together with dense connections. A benefit of this *Broadcast Module* is the transmission of the extracted feature maps to the higher-level layers such that the decision of the CNN is made based on all the extracted feature maps of the neural network. Furthermore, "it aligns with the intuition that audio information should be perceived at various time-frequency scales simultaneously" [92].

On the *GTZAN* dataset, the current top-3 state-of-the-art works [94] [95, 96] use a huge amount of features. Moreover, they emphasize the acoustic features at the same scale as the spectral features. Each of them also uses multiple networks that are fused to give the desired result. For instance, [95] uses a CNN architecture and Support-Vector Machine classifiers. A CNN model trained on the spectral features and a Multi-Layer Perceptron trained on the other features are used by [96].

Finally, several works use 1D CNN architectures directly fed with the raw audio waveform. The deep ResNet 1D CNN of [97] beats other 1D CNN state-of-the-art works. However, it still has a significant gap (more than 15% on the *GTZAN* benchmark) compared to the state-of-the-art.

Work	Feature(s)	GTZAN	Ballroom
W. W. Y. NG & al. (2020) [94]	Timbre, rhythm, mel spectrogram, harmonic spectrogram, percussive spectrogram, CQT, scattergram, and transfer feature	96.5	-
L. Nanni & al. (2018) [95]	Spectrogram, harmonic spectrogram, percussive spectrogram, CQT, scattergram, statistical spectrum descriptors, rhythm, modulation frequency variance descriptor, temporal statistical spectrum descriptor, and temporal rhythm histograms	95.7	-
D. Ghosal & al. (2018) [96]	Mel spectrogram, MFCC, energy normalized chromagram, CQT, tonnetz, tempogram, averaged signal vector, and music transfer learning vector	94.2	93.8
C. Liu & al. (2019) [92]	Mel spectrogram	93.9	96.7
R. Yang & al. (2020) [98]	Spectrogram	92	-
C. Sènac & al. (2017) [99]	Spectrogram, short-term energy, zero-crossing rate, brightness, spectral flatness, spectral shannon entropy, spectral roughness, key clarity, and harmonic change detection	91	-
H. Tang & al. (2020) [100]	Mel spectrogram	90.5	95.71
Y. Yu & al. (2020) [101]	Spectrogram	90	-
S. Allamy & al. (2021) [97]	Raw audio	80.93	-

Table 3.2: State-of-the-art convolutional neural networks for the *GTZAN* and *Ballroom* datasets. The authors of the model and the paper are shown in the first column. The feature set used by the model is mentioned in the second column. The two last columns are the accuracies [%] obtained on the benchmarks datasets.

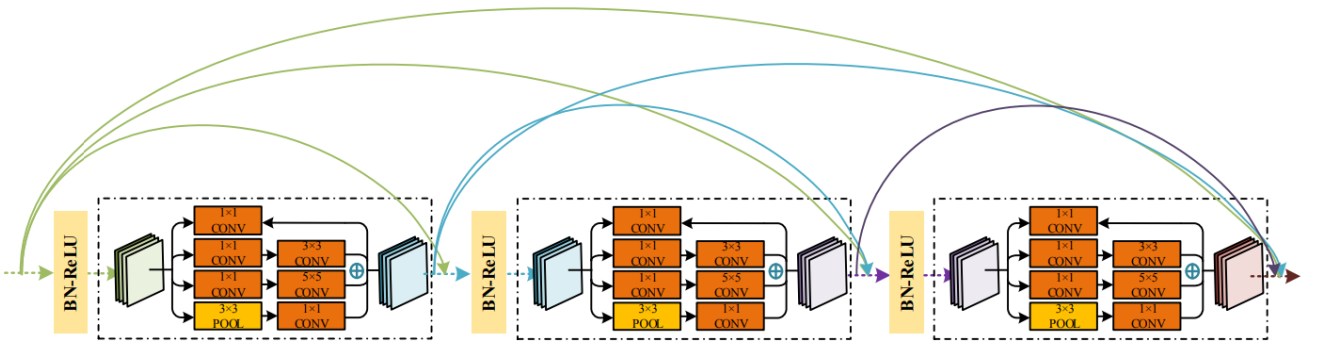


Figure 3.3: The *Broadcast Module* is composed of *Inception* blocks [93] stacked on top of each other and interconnected by dense connections [92].

3.3 Speech Classification

Speech Classification is one of the largest tasks in audio classification. It studies general voice recognition such as detecting the emotions in the voice, speaker identification, speech commands recognition, or still language detection.

3.3.1 Datasets

Enumerating all the datasets in Speech Classification would be unfeasible. Therefore, we consider describing the most famous benchmarks datasets for each of its subtasks.

The *RAVDESS* dataset [102] contains 7356 sounds, 4320 of them being speech recordings and 3036 song recordings. Twenty-four actors have made the records with 12 females and 12 males. Each person made 60 spoken utterances and 44 song utterances. Each of them was then exported into three distinct formats: audio-video, video, and audio. Each speech recording was realized with eight different emotions: calm, happy, sad, angry, fearful, surprise, and disgust, while song recordings were created using six different emotions from the actors: calm, happy, sad, angry, and fearful. The actors performed each vocalization twice at two levels of emotion (normal and strong). The audio samples have been recorded in a professional studio in uncompressed format at a sample rate of 48kHz and a 16-bit resolution.

The *Gender Recognition by Voice* dataset has been released on Kaggle [103] and contains 3 168 male and female voice recordings. It aims to improve genre recognition by speech analysis. The *VoxCeleb* dataset [104] is a large dataset studying speaker recognition by voice analysis. The dataset contains 153 516 voice recordings among 1251 celebrities. 55% of the recordings come from the male gender, making the dataset balanced. The audio samples have been obtained through Youtube. Therefore, the recordings vary a lot in terms of several aspects: ethnicity of the speaker, age, speed, profession, etc.

The *Google Speech Commands V1* dataset [105] is a collection of 65 000 1-second recordings of 30 short words. Google has recorded it with thousands of people included in the project and aims to improve keyword spotting applications. The *Google Speech Commands V2* dataset [106] is a second version containing 105 829 1-second recordings of 35 short words. Each sample has a sampling rate of 16 kHz and a resolution of 16 bits.

The *CSS10* dataset [107] is composed of audio and text speech recordings for ten different languages. The audio clips have been obtained through LibriVox audiobooks.

3.3.2 Related works

We focus on state-of-the-art of keyword spotting applications. As mentioned in [108], most CNN-based implementations for this speech recognition task use the *MFCC* feature. Table 3.3 highlights this fact with more than half of the state-of-the-art works using the MFCC feature. The best accuracy for the *Google Speech Commands V1* benchmark is achieved with this feature type [109]. This result comes from the *MatchBoxNet* architecture composed of 1D time-channel separable convolutions. These layers are similar to 2D depthwise separable convolutions and reduce the model size compared to 1D convolutions.

Work	Feature(s)	Google Speech Commands V1	Google Speech Commands V2	Thousands model parameters
Y. Wei & al. (2021) [110]	LFBE-delta	-	98.05	1720
S. Majumdar & al. (2020) [109]	MFCC	97.48	97.63	177
S. Mittermaier & al. (2020) [111]	Raw audio	96.4	97.3	62
D. Coimbra de Andrade & al. (2018) [112]	Mel spectrogram	95.6	96.9	202
D. Peter & al. (2021) [113]	Raw audio	-	95.55	75.7
S. Choi & al. (2019) [114]	MFCC	96.6	-	305
A. Soliman & al. (2021) [115]	MFCC	96.19	-	-
R. Tang & al. (2018) [116]	MFCC	95.8	-	238
Y. Zhang & al. (2018) [6]	MFCC	94.4	-	38.6
Y. Zhang & al. (2018) [6]	MFCC	91.6	-	79

Table 3.3: This table presents state-of-the-art convolutional neural networks for the *Google Speech Commands V1* and *Google Speech Commands V2* datasets with 12 words to be classified (a subset of the complete datasets). The authors of the model and the paper are shown in the first column. The feature set used by the model is mentioned in the second column. The two following columns are the accuracies [%] obtained on the benchmarks datasets. The last column is the number of parameters of the models in thousand units.

Compared to other categories in audio classification, the feature types used in the keyword recognition state-of-the-art are always the same: *MFCC*, *mel spectrogram*, or *raw audio*. Furthermore, CNN implementations that directly use the raw audio as input give impressive results in that field and are on the top of the state-of-the-art. An accuracy of 96.4% on *Google Speech Commands V1* and 97.3% on *Google Speech Commands V2* is achieved by [111]. They use a specific architecture represented in Figure 3.4. The first element is a SincConv layer. It restricts the filters of the 1D convolution layer to learn parametrized sinc functions. These sinc kernels require only two parameters, which reduces the memory footprint. Convolutioning the audio waveform with the sinc filters allows extracting parts of the recording in specific frequency ranges (learned by the filters). The other layers are depthwise separable convolutions.

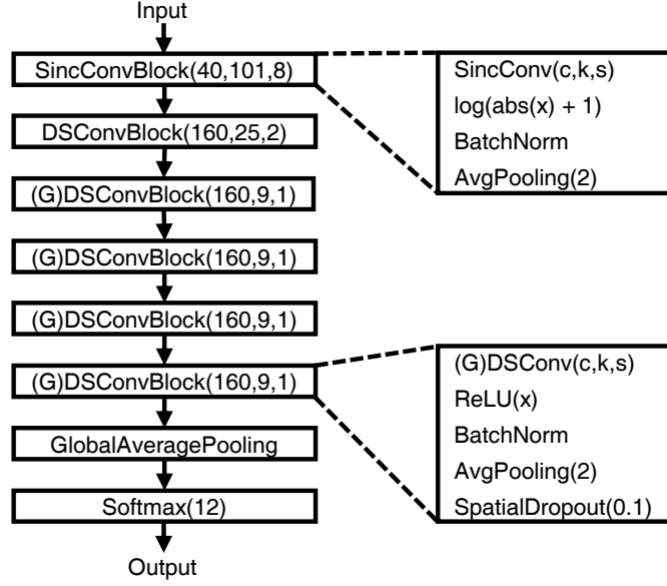


Figure 3.4: Novel architecture used by S. Mittermaier & al. It is composed of a SincConv layer and depthwise separable blocks [111].

Keyword recognition seems to be an easier audio classification task with respect to the previous ones. We can see it in the last column of Table 3.3 that shows the number of parameters of the proposed models. Most of the works create small energy-efficient models embedded into resource-constrained devices while reaching an accuracy above 90%. For instance, high-accuracy models have been developed and embedded onto a Cortex-M7 core by [6]. They demonstrate an inference time of 12 ms for their worst-performing keyword recognition model.

Finally, several works using transformers have been published for keyword recognition tasks. They are the ones that yield state-of-the-art performance with an accuracy of 98.6% by [117] on the *Google Speech Commands V2 dataset*. However, we do not discuss them here since their architecture is different from a CNN.

3.4 Bird Classification

Bird Sound Classification is the task of identifying one or numerous bird species in a recording. It has experienced a real boom in recent years following the arrival, among other things, of the *BirdClef* challenges [118]. Another trend is bird detection, which consists of identifying if a bird (regardless of the species) is part of a recording.

3.4.1 Datasets

The four presented datasets below contain bird recordings and environmental noise without birds. We thus use them for bird detection tasks.

The *Warblr* dataset [119] is a dataset of 10 000 10-seconds recordings from around the United Kingdom for a total duration of 28 hours. It has been recorded through the Warblr smartphone application, which is a crowdsourcing research project. [119] states that "the recordings cover a wide distribution of UK locations and environments, and include weather noise, traffic noise, human speech, and

even human bird imitations. It is directly representative of the data that is collected from a mobile crowdsourcing initiative. Annotations of the Warblr dataset are performed by a network of volunteers."

The *BirdVox-full-night* dataset [120] contains 35 402 sounds that were recorded by a network of six sensors during a night of birds migration around Ithica in 2015. This dataset has been processed for the DCASE challenges and has given life to the *BirdVox-DCASE-20k* dataset. It provides 20 000 10-seconds sounds at a sampling rate of 44.1 kHz.

The *freefield1010* dataset [121] is composed of 7690 10-seconds recordings at a sample rate of 44.1 kHz and a resolution of 16 bits. The audio samples come from the *Freesound* project. Therefore, the recordings are very diverse in location and environment.

The *PolandNFC* dataset is a set of 4000 recordings collected in Poland. More specifically, the collection is based on the Ph.D. project of Hanna Pamuła, and the samples were recorded every night for three months in 2016 on the Baltic Sea coast.

For bird species classification, the *Xeno-Canto* database is used. It is a huge collection of bird sounds from all around the world [122]. Everyone, being a professional or an amateur, is allowed to record a bird sound and upload it on the platform, contributing to wildlife conservation. More than 630 000 sounds have been uploaded to this day, and the database keeps on growing every day. This platform is well known and has been used extensively to provide bird datasets for bird species classification tasks such as all BirdClef competitions [123, 124, 125].

3.4.2 Related works

The DCASE 2018 challenge [126] has used the *Warblr*, *BirdVox-DCASE-20k*, *freefield1010*, and *PolandNFC* datasets for a bird detection task. The best result of this challenge is [127]. Interestingly, it uses a deep convolutional neural network pre-trained on the *ImageNet* benchmark, which is fine-tuned with *log-mel spectrogram* input images obtained from the audio. Furthermore, a lot of audio augmentation techniques are performed.

More works focus on bird species audio classification with a large number of species. However, no benchmark dataset exist since they are modified and enlarged every year at the occasion of the BirdClef challenges. We decide to review the three last challenges and their state-of-the-art implementations. We do not mention the results since they use other metrics not presented in this work.

The BirdClef 2018 challenge aims to classify 1500 species of bird from central and south America. The dataset contains 36 496 bird recordings with a large class imbalance. The best result is achieved by [128], which is the same winner as the DCASE 2018 challenge. He used an ensemble of seven deep CNNs pre-trained on *ImageNet* and fine-tuned on *mel spectrogram* or *log-spectrogram* images. The majority of the models are based on the Inception-V3 architecture [129]. Each model takes a different version of the feature type to feed the CNN. As before, it uses many audio augmentation techniques.

The purpose of the BirdClef 2019 challenge is to identify 659 bird species in soundscape recordings. The best implementation is again obtained by Mario Lasseck [130]: he uses an ensemble of deep CNNs with the Inception-V3 and ResNet architectures pre-trained on *ImageNet* and fine-tuned on

5-seconds *log-mel spectrograms*. He still performs plenty of audio augmentation techniques that yield significant improvements.

The training data used for the BirdClef 2020 challenge contains 70 000 recordings among 960 bird species. The objective of the challenge is the same as the previous year. It thus aims to classify numerous bird species in continuous soundscapes. The winner of this edition uses a DCNN directly operating on the raw audio waveform [131]. They perform a neural architecture search (NAS) to find their deep convolutional neural network architecture. Interestingly, their architecture is based on a 1D Gabor Wavelet layer. They also use intensive data augmentation. However, it is essential to mention that even though this work is the winner, bird classification in soundscapes recordings is far from being solved because the results are not performant at all.

3.5 Summary

This chapter presented the benchmark datasets and the state-of-the-art convolutional neural network implementations in four audio classification tasks: Environmental Sound Classification, Music Classification, Speech Classification, and Bird Classification. The three first tasks can be considered solved since the accuracy obtained on the benchmark datasets is extremely high (more than 96%). However, Bird Classification is more challenging and is intensively studied nowadays. Each task does not use the same feature types and the same neural network architectures:

- **Environmental Sound classification** typically uses the *log-mel spectrogram*, *log spectrogram*, and *CQT* features even though more of them are studied nowadays. The state-of-the-art CNNs are complex in practice, which yields non-energy-efficient models. To that purpose, more works focus on creating energy-efficient networks still able to produce high accuracy. Networks taking as input raw audio are such an example.
- **Music Classification** reached state-of-the-art performance with the *mel spectrogram* and *spectrogram* features. Today, it combines a lot more features, emphasizing acoustic features. In terms of networks, the audio-based models are not present, as they yield a big gap in performance compared to the state-of-the-art. Instead, multiple deep CNNs are fused to make the prediction.
- **Speech Classification** is considered a much easier task. Many energy-efficient convolutional neural networks are present in the state-of-the-art ranking, yielding outstanding results (more than 95%). Furthermore, audio-based networks are studied a lot since they reduce the model size while reaching state-of-the-art performance. For the image-based models, the *MFCC* and *mel spectrogram* features are the most used in practice.
- **Bird Classification** is a challenging task governed by an ensemble of deep neural networks using most often the *log-mel spectrogram* feature. Studying energy-efficient models in this task is not expected for many years because even deep neural networks have trouble solving the job efficiently.

Forest audio classification is the task we target in this Master thesis. We want to detect animal, bird, chainsaw, and gunshot sounds with an ultra-low-power model. As seen in this chapter, it has never been done in the literature. However, it can be considered as a subtask of Environmental Sound classification. We are thus aware of the state-of-the-art spectral features of this field and the promising advances in energy-efficient models.

In this chapter, we analyze the original gathered dataset and explain the sequence of pre-processing operations used to obtain our actual dataset, referred to as the *Events* dataset. Chapter 5 divides the latter in training, validation, and test sets to train a CNN. The *Events* dataset contains animal, bird, chainsaw, and gunshot recordings to solve the targeted classification task.

4.1 Gathering of the original dataset

Obtaining a relevant audio dataset for a classification job is not an easy task. It is indeed more difficult than getting an image-based dataset for an image classification project. The latter is well-known in the scientific community and has been a field of research for a few years, while audio classification is less common. Finding a good dataset for our audio classification task to distinguish between chainsaw, gunshot, bird, and animal sounds is impossible only with the Internet. Indeed, the 4-classes audio classification task we solve in this work has never been done in the literature. As mentioned in Section 3.1.1, there are a few well-known audio datasets for Environmental Sound Classification, but they do not fit the target task, or they do but with a small amount of training data.

Two enterprises in the United States have been contacted to obtain the required data. The first one, called *Rainforest Connection* [132], was founded in 2014 by Topher White to protect the rainforests in the world. Back at that time, he was an engineer on a mission on Borneo and saw large-scale illegal deforestation [133]. This phenomenon has a dramatic effect on global warming as well as on the life of animals and poorer communities [15]. However, it is difficult to prevent it because of the low number of forest rangers and the vast area to cover. It was too much for Topher White, who created and deployed new solutions to reduce deforestation worldwide. The main idea is to recycle old mobile phones and use them as IoT sensors by running machine learning models that can detect illegal chainsaw activities. These devices are hidden high in tree canopies (as depicted in Figure 4.1) and embedded with a powerful microphone able to capture sounds up to a 1km radius. Furthermore, the use of solar panels for long-term sustain makes them environmentally efficient. If the model which is embedded in the mobile phone detects the sound of a chainsaw, then an alert is directly sent, in real-time, to the forest rangers of the area. The diagram flow is represented in Figure 4.2. Nowadays, *Rainforest Connection* implants these devices in numerous countries, leading a global fight against deforestation. Furthermore, they also fight against animal poaching and contribute to wildlife conservation. The thousands of hours captured by these sensors allow biologists to study more in-depth the animals present in specific areas of the globe.



Figure 4.1: Topher White installing a Rainforest Guardian (sensor) high in the trees [15].

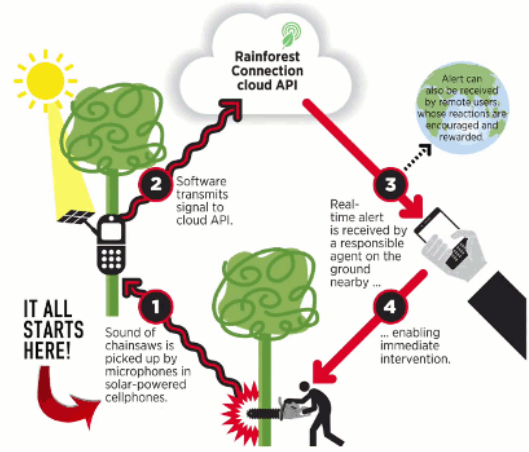


Figure 4.2: Diagram flow of *Rainforest Connection* mindset [133].

The second institution, the *Borror Laboratory of Bioacoustics* [134], is part of the Ohio State University. This museum possesses a rare collection of animal sound recordings, "one of only about ten collections of animal sound recordings in the world" [135].

Thanks to *Rainforest Connection*, a huge amount of data is obtained: they proposed that we participated in one of their internal challenges, for which the purpose was to create a convolutional neural network able to classify between chainsaw and environmental sounds. In that context, they agreed to give us a complete dataset coming from the forest field containing 88 567 chainsaw and 88 948 environmental sounds of 2 seconds each. As part of this thesis, they also decided to provide us with 4909 gunshot sounds coming from different sources:

1. From three different study areas in Greece: the Evros National Park, the Ródope Mountain National Park, and the Zante Island ;
2. From the field (recorded by *Rainforest Connection* guardians) ;
3. Generated and mixed with other sounds.

We contacted the *Borror Laboratory of Bioacoustics* to find an animal sounds database. After explaining our case, they agreed to grant us their entire collection of mammal recordings containing 1099 sounds.

The bird dataset is obtained online from *Xeno-Canto* [122]. Because of the large number of recordings needed for our audio classification task, it is impossible to retrieve all of them manually. We thus use a web-scraper, allowing us to download automatically, with the help of a Python script, audio sounds from *Xeno-Canto*. It is possible thanks to the wonderful web-scraper that can be found on Github [136]. Note also that a choice needs to be made concerning the dataset we are building from *Xeno-Canto*. We decide to download all bird recordings from Belgium (4822 samples) to have, later, a testable and deployable device in the Belgium forests.

Finally, we extend the gunshot dataset from *Rainforest Connection* with the *Gunshot Audio Forensics Dataset* because of the overall lack of gunshot recordings. This last dataset was collected in Summer 2017 in rural Arizona. Twenty firearms were recorded from twenty different positions on four different

recording devices. However, we decide to keep only the recordings from one device (1990 samples) not to have too much similar audio samples. The complete information about the dataset can be found here [137]. The *Gunshot Audio Forensics Dataset* data collection was supported by NIJ grant 2016-DN-BX-0183 “Development of Computational Methods for the Audio Analysis of Gunshots.”

The number of audio samples obtained by each organization and for each class is summarized in the following table. Note that the environmental class is not part of the classification task we target but instead used for data augmentation, as discussed later in Section 5.3.3.

	Animals	Chainsaws	Gunshots	Birds	Environmental
<i>Rainforest Connection</i>	0	88 567	4909	0	88 948
<i>Borror Lab.</i>	1099	0	0	0	0
<i>Xeno-Canto</i>	0	0	0	4822	0
<i>Gunshot Audio Forensics Dataset</i>	0	0	1990	0	0

Table 4.1: All audio datasets have been obtained from four organizations.

4.2 Analysis of the original dataset

As shown in Table 4.1, there exists a massive imbalance in the number of recordings for each class. Furthermore, audio recordings were recorded in various conditions, requiring careful investigation to harmonize our actual dataset. Multiple analyses are thus conducted to show the diversity of the original dataset. These analyses rely on multiple Python packages such as `librosa`, `matplotlib`, `numpy`, `opencv-python`, `scipy`, and `ffprobe`.

4.2.1 Sample rate

Figure 4.3 shows the sample rate repartition among the different classes. Two conclusions arise:

1. The sample rate between different audio recordings changes a lot in the range $[8 ; 48]$ [kHz] ;
2. The sample rate is constant for the audios from the animals and chainsaws classes with a value of respectively 48 kHz and 12 kHz. It is because these two are already prepared datasets coming from professional enterprises. The birds class comes entirely from *Xeno-Canto*, in which every bird amateur can upload its own recordings. On the other hand, the gunshot dataset has been obtained by multiple sources, explaining the different sample rates depending on the recording devices.

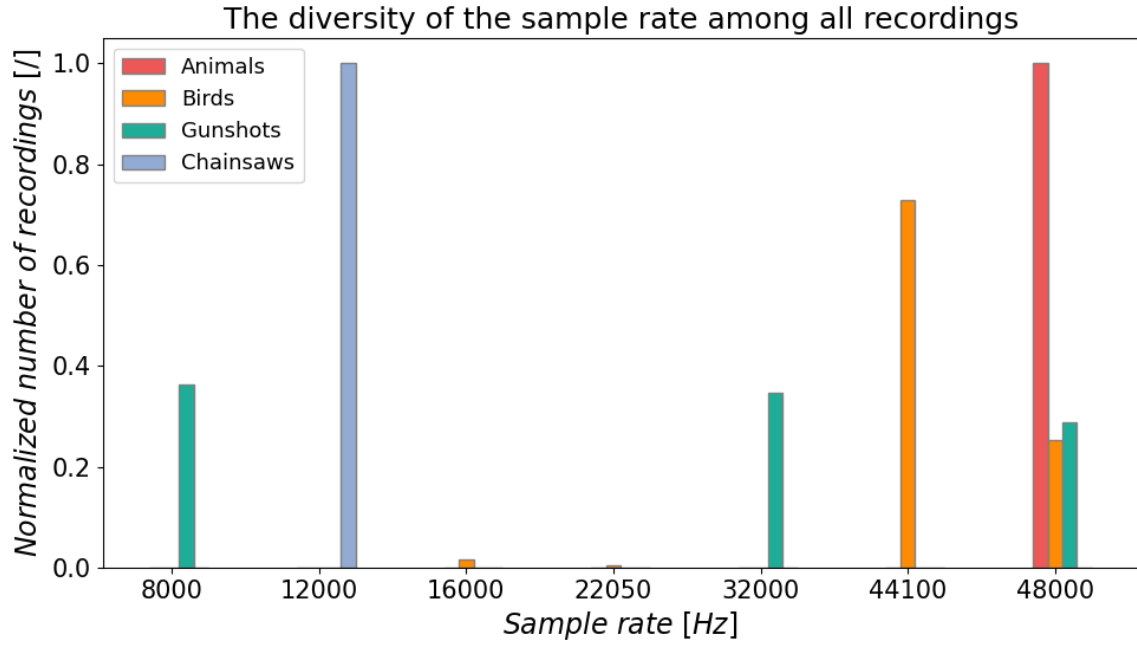


Figure 4.3: The sample rate is not constant. The sample rate of an audio from the original dataset can take one of the following values [8 ; 12 ; 16 ; 22.05 ; 32 ; 44.1 ; 48] [kHz].

4.2.2 Number of audio channels

Figure 4.4 shows the audio channels repartition of the recordings among the different classes. From it, two conclusions, similar to the ones for the sample rate analysis, are drawn:

1. The number of audio channels between different audio recordings changes in the range [1 ; 2] ;
2. The number of audio channels is constant for the audios from the animals and chainsaws classes with exactly one audio channel. The reason is the same as the one proposed for the sample rate analysis in Section 4.2.1.

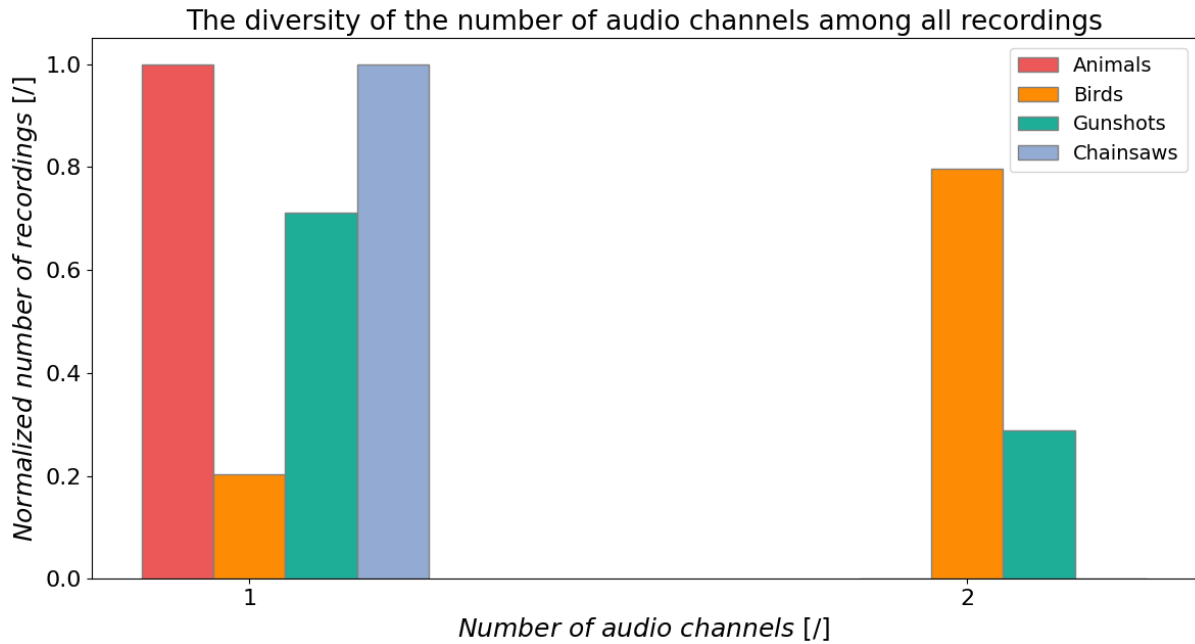


Figure 4.4: The number of audio channels is not constant depending on the recording. The number of channels of an audio from the original dataset can take one of these values [1 ; 2].

4.2.3 Duration of the audio recordings

Figure 4.5 presents histograms of the duration of the recordings for each class. As before, we can see that *Rainforest Connection* has well designed the chainsaw dataset. Each recording in this dataset lasts exactly 2 seconds, in accordance with the dataset purpose : create a deep learning model able to detect chainsaw 2-seconds sounds. Gunshot recordings last between 1 and 4 seconds, while bird and animal recordings go much higher in duration.

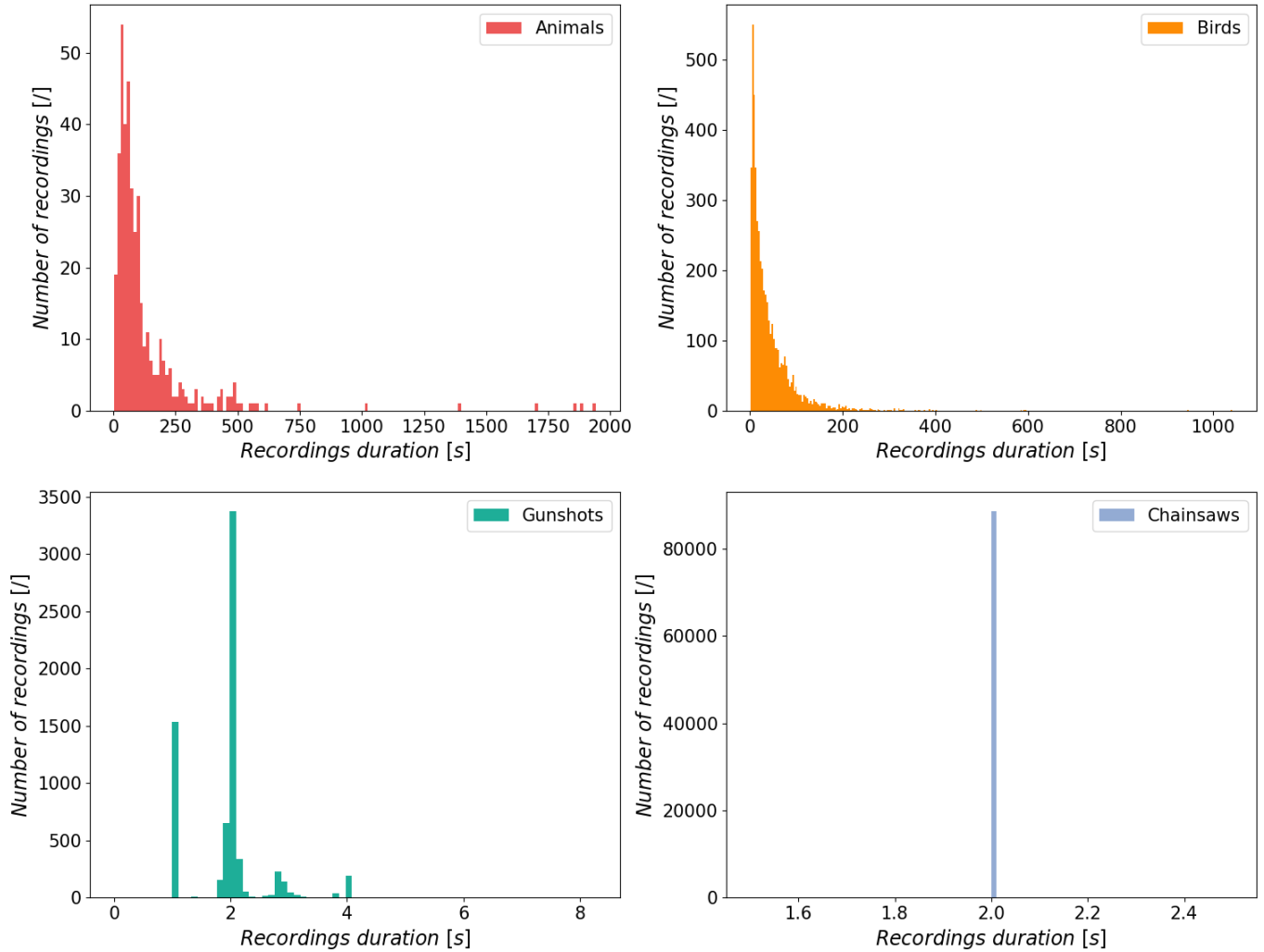


Figure 4.5: Recordings duration histogram for each class. The duration of the audio samples depends a lot on the class.

4.2.4 Resolution of the audio recordings

Table 4.2 shows the resolution (or bit-depth) of the recordings of each class. Bit-depths are identical except for the bird dataset, for which the information cannot be found.

	Animals	Chainsaws	Gunshots	Birds
Resolution	PCM signed 16 bits	PCM signed 16 bits	PCM signed 16 bits	Unknown

Table 4.2: The different datasets do not have the same resolution.

4.2.5 Quality of the audio recordings

Unfortunately, as shown in Table 4.3, the bird dataset is compressed with the mp3 codec. The reason is that *Xeno-Canto* essentially allows mp3 recordings, and the uploaders do not see the advantage of using an uncompressed format. However, the uncompressed audio format is far more valuable in reality since "it provides an accurate copy of wildlife sounds. By recording in the WAV format, you will maximize the usefulness of your recordings for research and conservation both today and in the future. Due to the compression, these formats selectively delete certain acoustic information in a recording and this loss of information can affect the ability of researchers to examine acoustic properties of the sound files. When it comes to bird songs, the audio deletion can be misleading and problematic because birds hear sounds differently than humans. This means that audio that might be unimportant for human listening could be important for bird listening" as stated in [138].

	Animals	Chainsaws	Gunshots	Birds
Quality	Uncompressed (.wav)	Uncompressed (.wav)	Uncompressed (.wav)	Compressed (.mp3)

Table 4.3: The different datasets do not have the same audio quality.

4.2.6 Proportion of useful signal with respect to the original one

Each audio recording of the original dataset does not contain the sound of interest during its whole duration. For instance, for a bird recording, we want to have bird sounds. We are not interested in the sound of the rain, vehicles, etc., that could happen in the same labeled bird recording. This experiment aims to show, for each class, the proportion of relevant sound with respect to the entire duration of the recordings. Thus, there is a need for a segmentation technique allowing to separate the important sound, the signal, and the remaining one, the noise. The segmentation technique is presented in detail in Section 4.3.

Because this experiment uses the segmentation technique, we need beforehand to know its behavior. It outputs a temporal mask allowing to obtain audio segments of important sound and audio segments of noise sound. We dismiss the noise segments for this specific experiment and concatenate the signal segments to recreate one recording. Then, we divide the length of the segmented recording by the initial length of the recording. After doing so with every recording and with each class, we obtain four vectors (one per class) in which each element is the ratio of lengths corresponding to one recording. We can thus plot the kernel density estimate, estimation of the probability density function, of the ratio of lengths for each dataset. However, by doing so, it should be noticed that we obtain a loss of temporal resolution since we put together segments of important sound that are further away initially. For instance, two bird calls that are spaced by 5 seconds with noise in between become a small sound where the two calls are made just one after the other. The temporal resolution is therefore lost. Fortunately, this is acceptable for this experiment since we want to know the proportion of important sounds with respect to the original audio sample.

The kernel density estimate of the proportion of useful signal with respect to the original one for each class is shown in Figure 4.6. Note that the chainsaw estimate is not represented since all recordings contain only important sounds. It means that each 2-second chainsaw recording contains a sound of a chainsaw for 2 seconds.

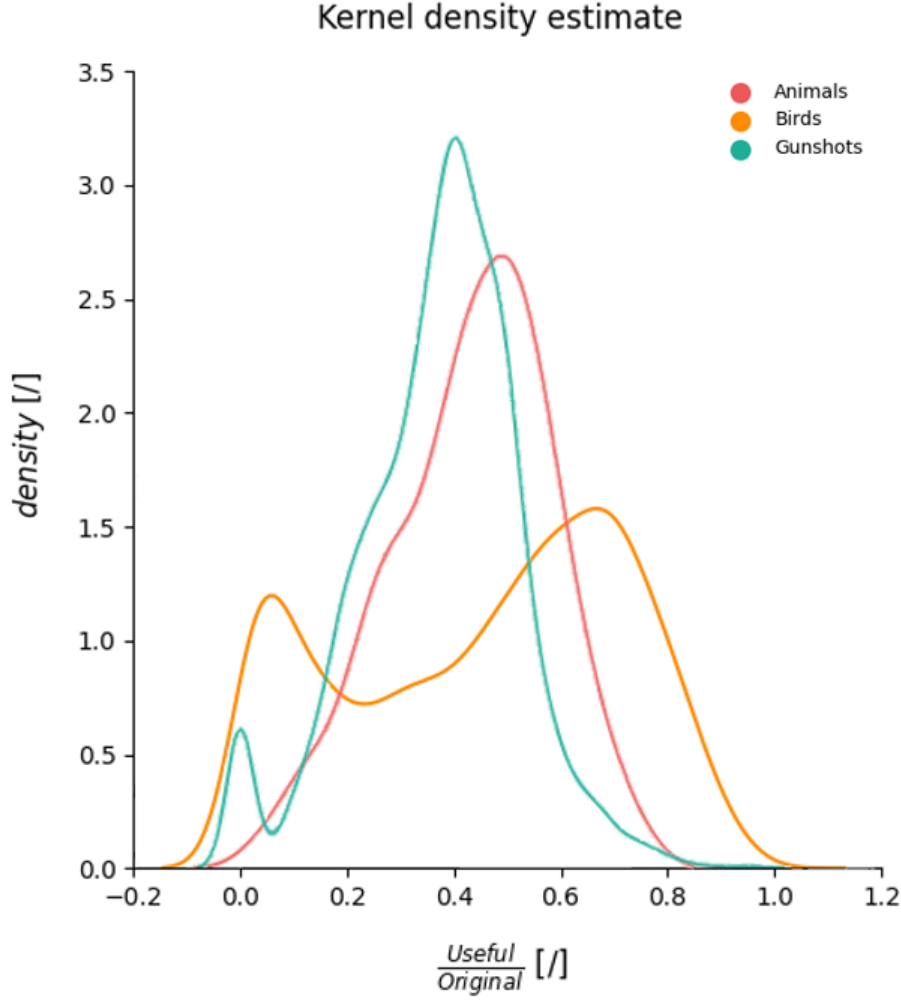


Figure 4.6: Kernel density estimate of the proportion of useful signal with respect to the initial one for each class. The chainsaw dataset contains only interesting sounds, reason why the kernel density estimate of this class is not represented. The values **-0.2** and **1.2** on the x-axis are due to the inner working of the KDE method.

As we can see, none of the three datasets represented in Figure 4.6 contain only relevant sounds. This experiment proves that a segmentation of the recordings is needed not to have irrelevant sound presented to the convolutional neural network. For instance, a 10-seconds bird recording containing 5 seconds of bird calls and 5 seconds of a thunderstorm would give to the CNN thunderstorm sounds labeled as birds. However, this is a bad process since it could lead to the learning of unwanted features unrelated to the birds class.

4.2.7 Conclusions

The original dataset that has been collected needs to be processed because it contains data that do not have the same properties depending on the class it belongs to. A relevant dataset is a dataset for which the properties of the data are the same whatever the data sample. Moreover, the duration of the recordings is highly variable, and a fixed one is needed to train a deep-learning model. Finally, the datasets contain a lot of irrelevant noise that has to be segmented cleverly. The following section explains the segmentation technique that we use, and the next one studies the choices made to obtain a relevant dataset.

4.3 Segmentation Technique

The segmentation technique is based on the work done by Mario Lasseck [139] and has proven to be very effective in numerous BirdClef challenges [140, 128]. This technique is due to the fact that original audio recordings often do not contain the important sound all the time but only a fraction of the entire recording. Thus, the idea is to be able to segment the signal part from the noise part in each recording. The signal segments would be kept and reused later to create the *Events* dataset, while the noise segments could be used as background noise for data augmentation. We do not keep noise segments in this work since we use the environmental dataset from *Rainforest Connection* for the data augmentation.

The complete overview of the method is explained here, based on [140], whereas the parameters used in the technique come either from the baseline [140] or by experimentation.

1. We compute the spectrogram of the sound. For the short-time Fourier Transform, a Hanning window of 512 samples and 75% overlap is used ;
2. The spectrogram is normalized in the interval $[0 ; 1]$ by subtracting the minimum value of the spectrogram to each element and dividing each new element by the initial maximum value of the spectrogram ;
3. The normalized spectrogram is passed through a median blur, a specific kind of filter known to remove the salt-and-pepper noise (sparsely occurring white and black pixels) effectively with considerably less edge blurring than other linear filters. The principle of the median blur is that "it takes a median of all the pixels under the kernel area and replaces the central element with this median value. Because we are taking a median, the output image has no new pixel values other than those inside the input image." [141]. This step is not used in the baseline technique ;
4. In the processed spectrogram, we select all pixels that are $4.5\times$ bigger than the row and column median. Intuitively, if we consider that a high amplitude usually corresponds to a signal of interest, this gives all the important parts of the spectrogram. The selection means that we set the pixels that respect the conditions to 1 and everything else to 0. The output of this step is thus a binary image ;
5. An opening of the binary image is performed. The purpose is to remove some of the foreground (bright) pixels from the edges of regions of foreground pixels. We thus want to remove with a second pass the salt noise in the image. Concretely, an opening is a morphological operation that consists of two consecutive other morphological operations : erosion and dilation. It allows, in that specific order, to remove the salt noise in an image. Experimentally, we found that a 4×4 filter produces the best results, as demonstrated in [140].
6. We create a vector in which the number of elements equals the number of columns in the spectrogram. We set to 1 the i -th element in this vector if the i -th column contains at least one 1. Otherwise, it is set to 0 ;
7. The vector is smoothed by applying two binary dilation filters (filter size 4×1) ;
8. We scale the vector to the length of the original audio file to use it as a mask allowing the extraction of the signal part from the entire audio. The output mask with a positive value involves a signal part, while a null value implies a noise part.

To have a deep understanding of the method, we represent the sequence of operations for a dog audio file from the original animal dataset. We show it twice : once with the median blurring operation and once without it to see its benefits. However, note that the second step (the normalized spectrogram) is not shown since it is visually the same as the first step. We also present a block diagram summarizing each step in Figure 4.9.

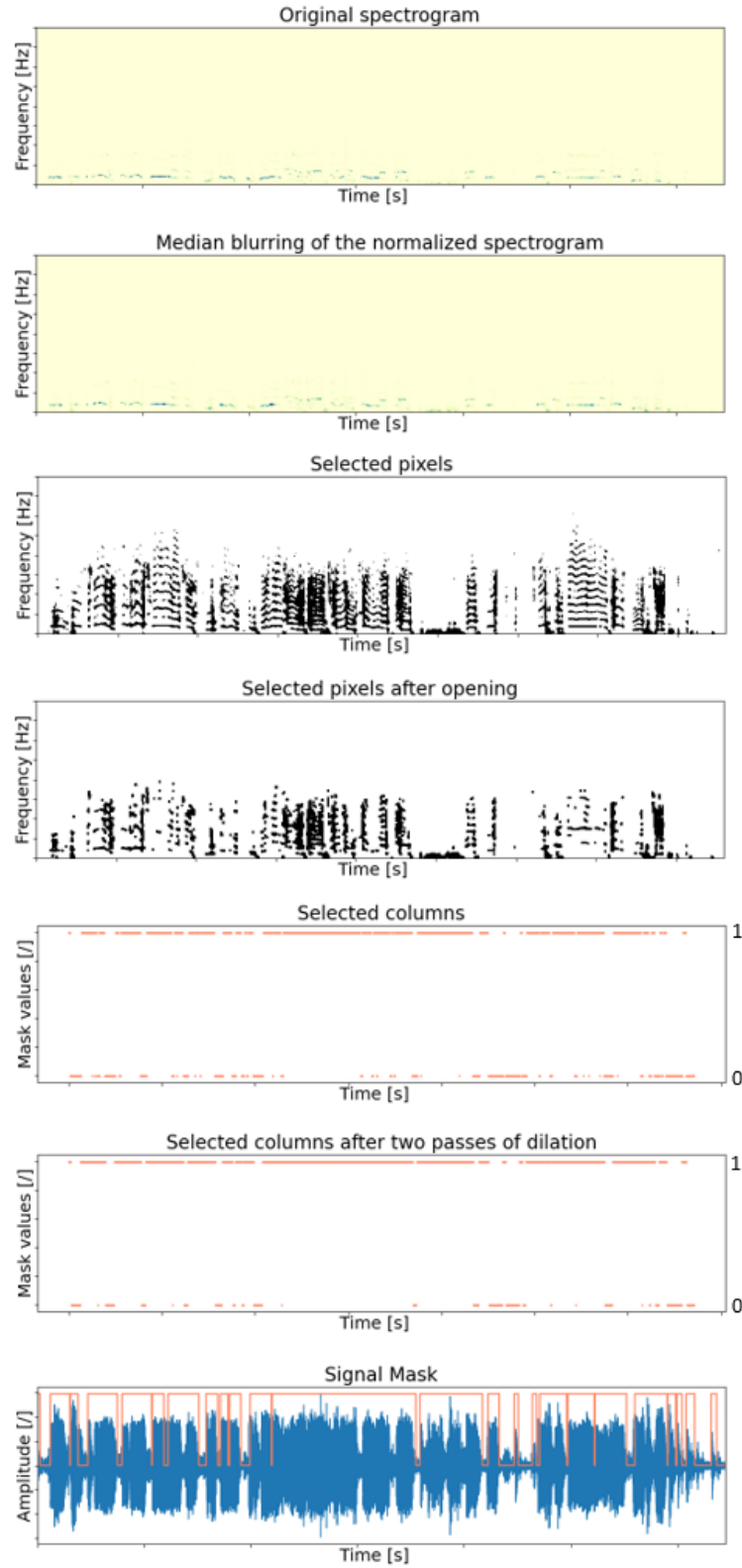


Figure 4.7: Segmentation technique sequence of operations with the median blurring operation.

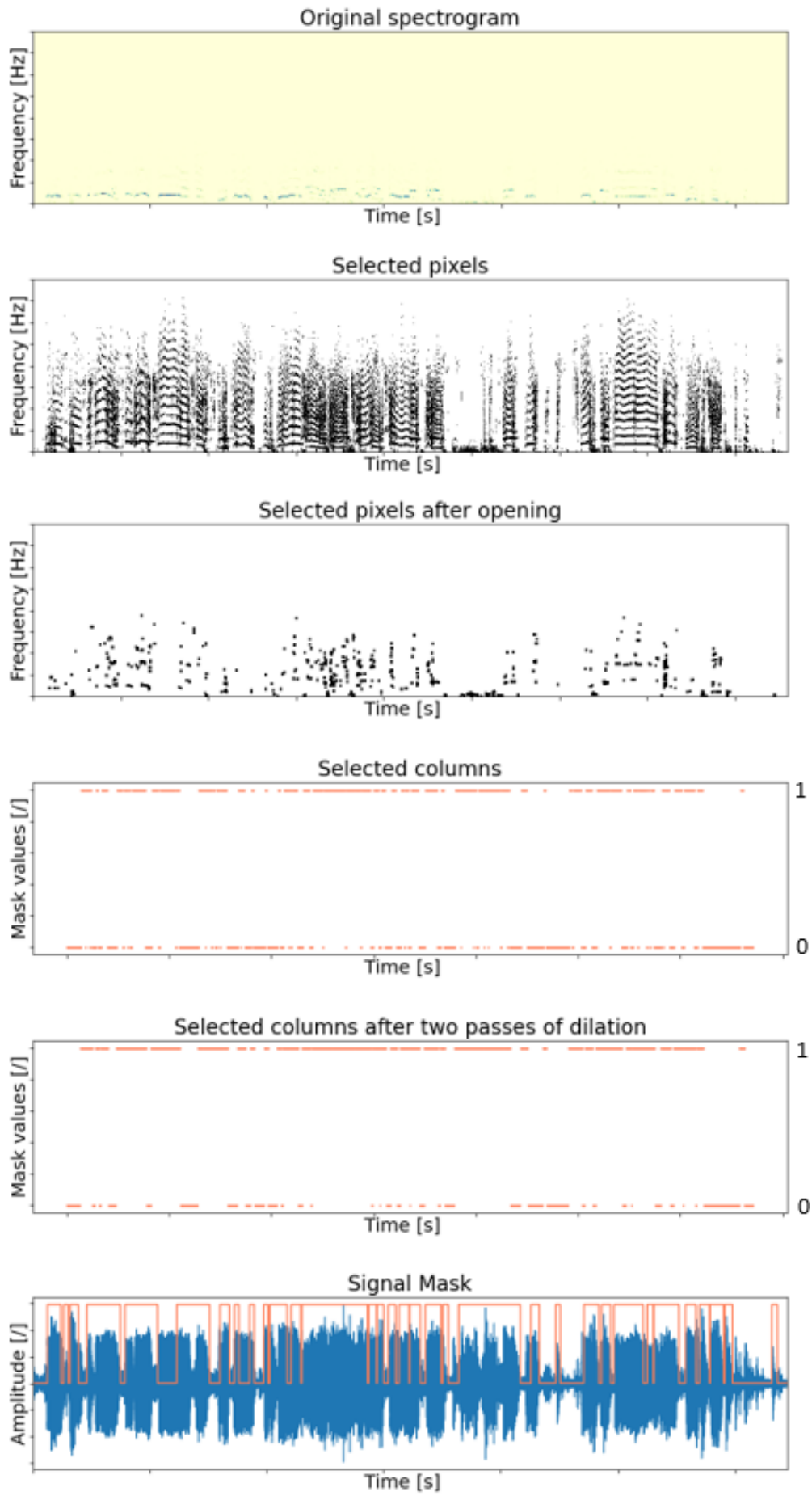


Figure 4.8: Segmentation technique sequence of operations without the median blurring operation.

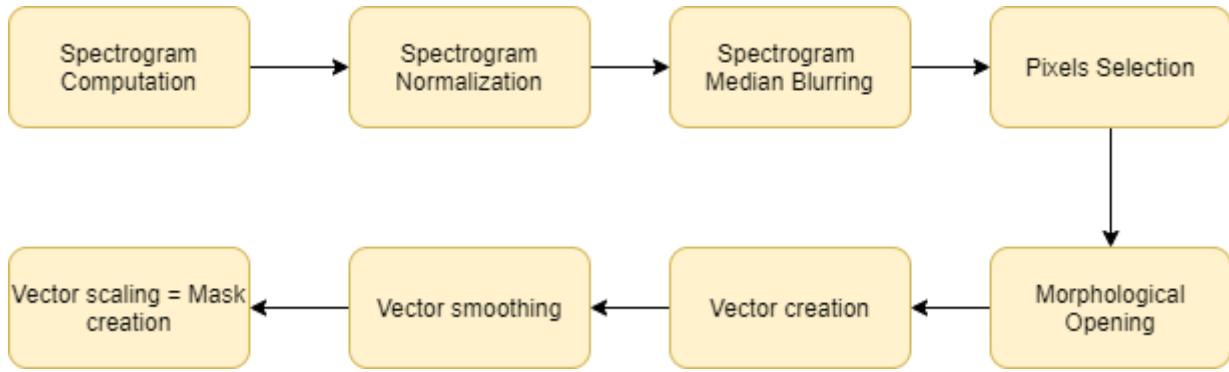


Figure 4.9: Block diagram of the segmentation technique.

We see that the median blurring operation in Figure 4.7 allows removing salt and pepper noise in the normalized spectrogram. It gives a cleaner binary image (in the **Selected pixels** step) before the opening operation. This blurring operation has a minimal theoretical effect in reality since the opening operation is supposed to remove noise. Still, experimentally, the pipeline with the median blurring operation yields better results.

Nonetheless, we make one important assumption with this technique: we consider a quiet environment where the sound of interest is higher in intensity than the background noise. If we had a noisy environment, we would consider as signal segments parts of the recording which are not relevant, like the rain, the wind, a fan, etc. However, this is, of course, not always true, leading to a sub-optimal and non-perfect segmentation technique even though the results are encouraging. Note also that the segmentation technique is relatively easy to program. The Python code needed to implement the whole segmentation technique is shown in Appendix A.1 and is based on [142].

There exists a wide range of possible segmentation techniques. For instance, we could want to detect signal parts by selecting time windows with an energy density bigger than $X\%$ (with X a threshold) of the average energy density of the audio signal. The idea is represented in the following figure.

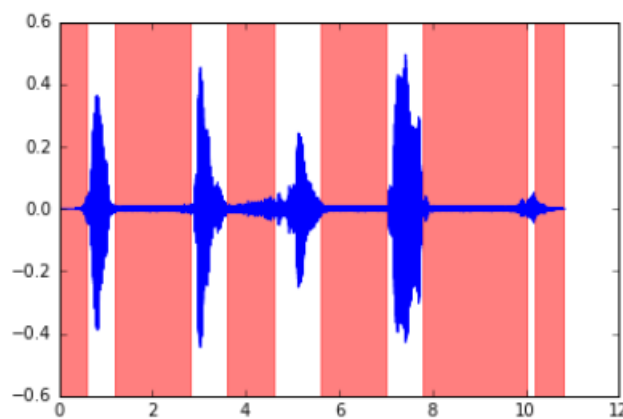


Figure 4.10: Regions shaded in red are not taken (noise parts), while areas let in white are taken (signal parts). The author of the work chose 4% for the threshold since it gave good results. The audio file is a cat sound [143].

This technique is implemented and used by [143] for an animal sound recognition project.

4.4 Compilation of the *Events* dataset

In this section, we discuss each of the steps and choices made to process the original dataset. We sequentially address issues and the engineered solution. Each section uses as dataset the one obtained in the previous step to finally converge to the *Events* dataset.

4.4.1 Animal dataset filtering

The original animal dataset contains 1099 sounds of mammals from many different species recorded in the United States. However, a lot of these sounds involve animals that are not living in forest environments. For instance, the dataset contains whale sounds, lion sounds, and others. Since this work aims to enable the use of a device for forest monitoring, it is irrelevant to keep these out-of-the-aim species. As a result, we only keep 410 animal sounds.

4.4.2 Optimal sampling rate choice

As highlighted in Section 4.2.1, the sample rate changes depending on which class the recordings belong to. However, to build a good dataset for a deep-learning algorithm, we must choose the same sampling rate for each category. An important step here consists of finding the optimal sample rate given our data and application constraints.

Because we look for an ultra-low-power model, we must choose the sample rate as small as possible since it directly impacts the input size of the image (the larger the sample rate, the larger the input image). It is due to the short-time Fourier Transform behavior, as shown in Equation 2.14. A bigger input size for the network involves increased inference latency and energy because more computations have to be done.

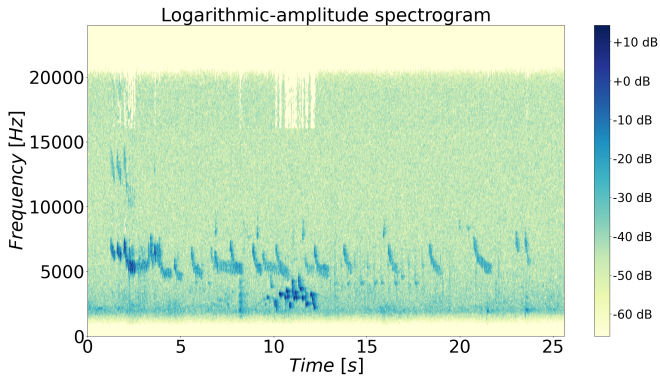
However, we know that the sample rate also defines the Nyquist frequency :

$$f_s = \frac{SR}{2} \quad (4.1)$$

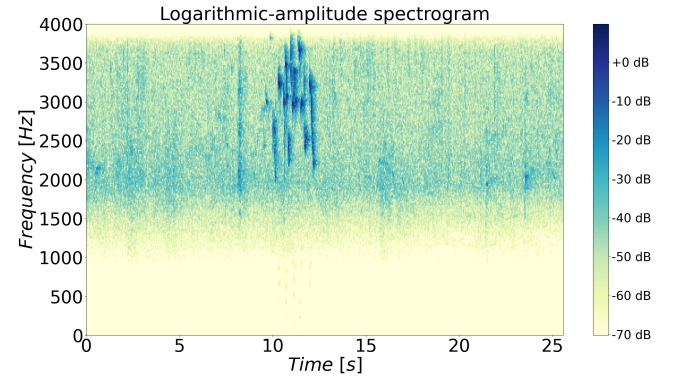
where f_s is the Nyquist frequency and SR the sample rate.

The Nyquist frequency is the maximal frequency that we can analyze in a signal. Therefore, it yields a trade-off between signal information, the key to a successful classification, and processing efficiency.

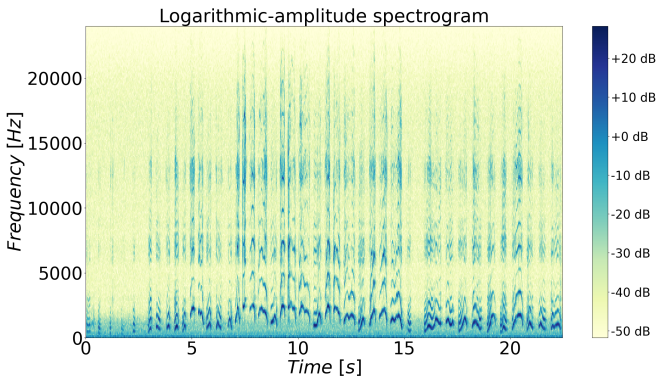
Thus, the question is to obtain the smallest sample rate that can minimize the loss of information regarding the frequency content of the complete dataset. Fortunately, we must only do this analysis for the animal and bird datasets since the chainsaw and gunshot signatures are still recognizable at a much lower sample rate. Figure 4.11 proves this reasoning by showing the logarithmic-amplitude spectrogram of a recording of each class at a sample rate of 48 kHz and 8 kHz (it is possible to change the original sample rate to another by upsampling or downsampling with the `librosa` Python package). The logarithmic-amplitude spectrograms have been obtained using a short-time Fourier Transform with the following parameters: a Hanning window of 512 samples and 75% overlap.



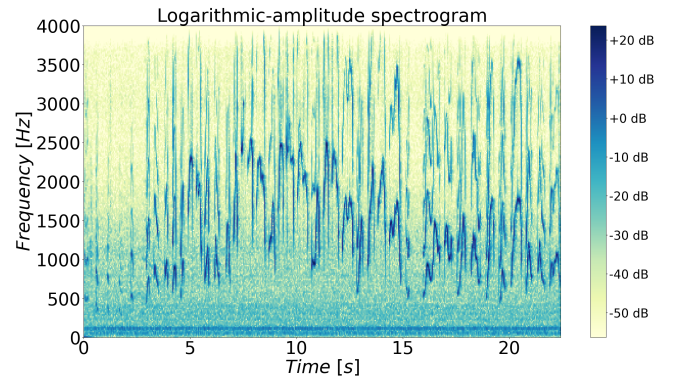
(a) Bird recording at 48 kHz.



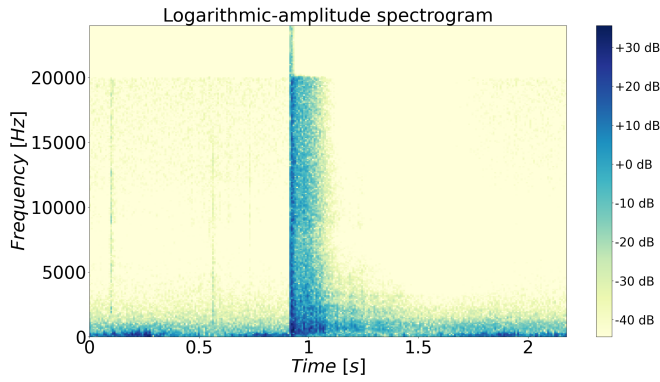
(b) Bird recording at 8 kHz.



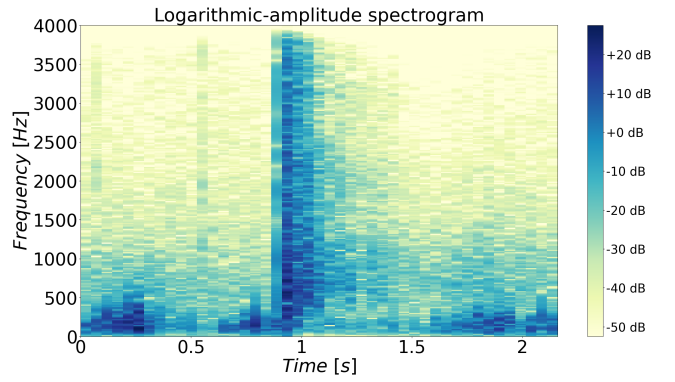
(c) Animal recording at 48 kHz.



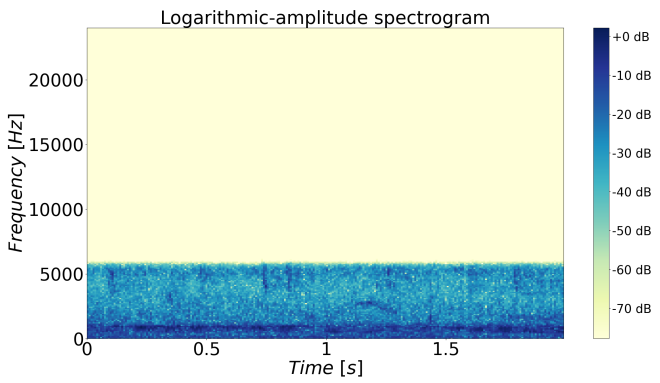
(d) Animal recording at 8 kHz.



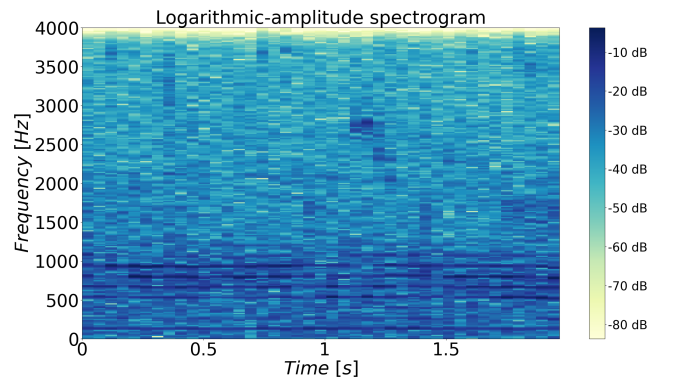
(e) Gunshot recording at 48 kHz.



(f) Gunshot recording at 8 kHz.



(g) Chainsaw recording at 48 kHz.



(h) Chainsaw recording at 8 kHz.

Figure 4.11: Logarithmic-amplitude spectrogram of a recording of each class at a sample rate of 48 kHz and 8 kHz.

Figure 4.11 showcases that the signature is visible at the two different sample rates for the chainsaw and gunshot recordings, meaning that a much lower sampling rate does not severely impact these two datasets. However, for the bird and animal recordings, the result is different. We lose almost all visible spectrogram information by going from 48 kHz down to 8 kHz. Because the sample rate impacts only the bird and animal datasets, the technique used to find the optimal sample rate is only performed on these two datasets.

The chosen pipeline to find the optimal sample rate is presented hereafter. Note that the technique has to be performed on the animal and bird **segmented** datasets following the same setup as the one presented in Section 4.2.6. Thus, each segmented recording is obtained by concatenating together all signal segments from the mask output of the segmentation technique. This specific setup is required since we need to remove the noise parts of the recordings. Otherwise, they would contribute randomly to the pipeline because the power spectral density of the noise is flat.

1. We compute the logarithmic-amplitude spectrogram of the recording ;
2. For each temporal frame (= column of the spectrogram), we extract the maximum of the spectrogram ;
3. For each temporal frame, we find all frequential bins (= row indexes) that are bigger than or equal to the maximum of the corresponding column of the spectrogram minus 10 dB. It gives roughly all the important parts of the signal we consider. Note that the value **10 dB** is a hard threshold that has been found by trials and errors to give the best performance ;
4. For each temporal frame, we take the biggest index (because a low index corresponds to a low frequency while a big index corresponds to a high frequency) among the set of indexes from the last point. Then, this index can simply be converted back to a frequency. The end of this step allows thus to obtain the highest frequencies across the entire recording ;
5. We perform an exponential moving average filter on the output of step 4 since there are glitches in the process. It allows to reduce them. Such filtering is given by : $EMA_t = \alpha \times value_t + (1 - \alpha) \times value_{t-1}$. We choose $\alpha = 0.2$ because we tend to give more weight to the previous value. The reason is that there are only a few milliseconds between consecutive frames (if the hyperparameters of the short-time Fourier Transform are well-chosen), and we assume that a song or audio does not change significantly in frequency in this small lapse of time ;
6. We take the mean and the standard deviation of the highest frequencies of the whole recording. We add these two numbers together and add the result to a histogram ;
7. We do the same operations (from steps 1 to 6) for another recording of the dataset (different from those already processed) ;
8. When all recordings of a dataset have been processed, we plot the histogram and compute different percentile data.

The process is schematically represented in Figure 4.12.

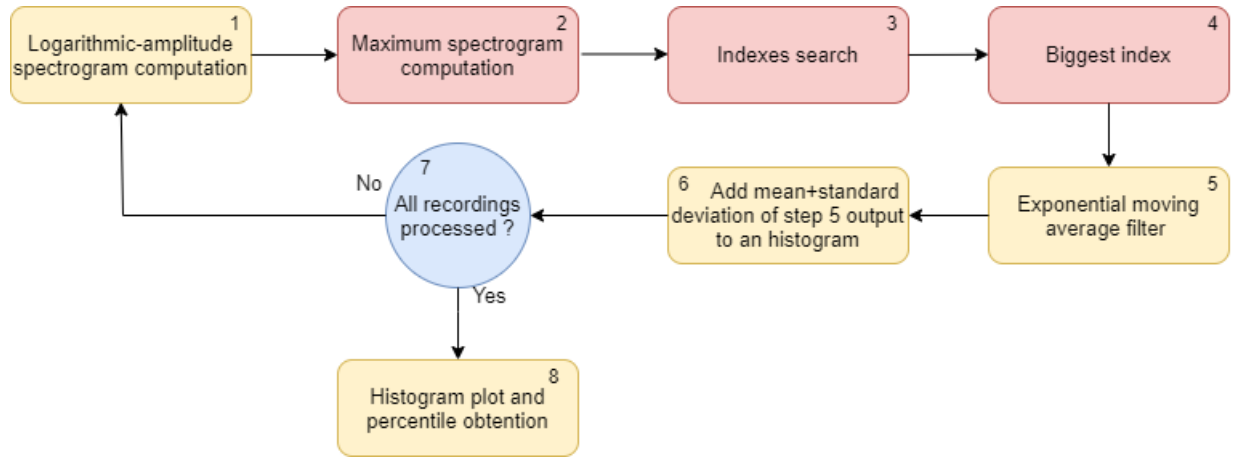
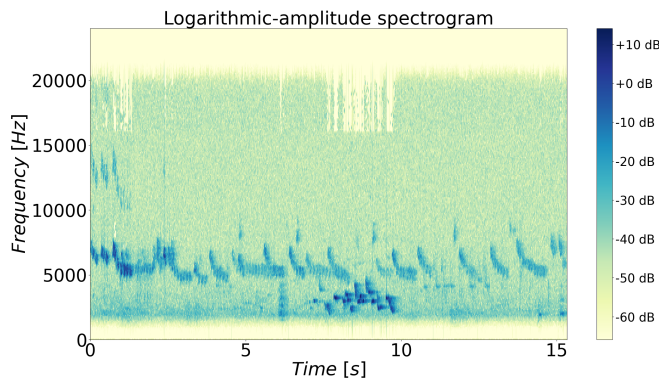
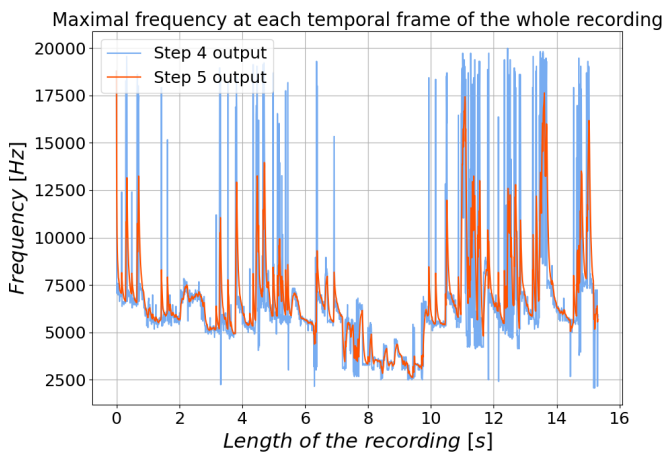


Figure 4.12: Pipeline to find the optimal sample rate with our dataset. The red rectangles represent steps that are done on each temporal frame of the spectrogram.

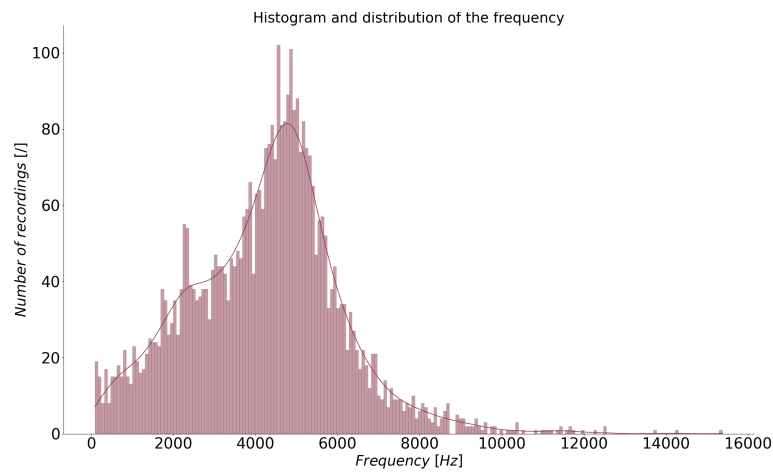
For quantitative illustration, we showcase in Figure 4.13 key step results in the extraction process for the same bird recording as in Figure 4.11a. However, the logarithmic-amplitude spectrogram in Figure 4.13a is not the same since we work here with the segmented recording.



(a) Logarithmic-amplitude spectrogram : end of step 1.



(b) The highest frequency of each frame for the whole recording: end of step 4 in blue ; end of step 5 in orange.



(c) Histogram for all bird recordings : end of step 8.

Figure 4.13: Optimal sample rate : important steps of the pipeline.

From the final histograms (by doing it for the animal dataset as well), percentiles can be found to have a deeper analysis of the problem. The most important ones are presented in Table 4.4.

	3th percentile	25th percentile	50th percentile	75th percentile	90th percentile	95th percentile	97th percentile
Animals	698	1676	2294	4094	6667	8362	10146
Birds	1259	4019	5536	6646	7892	9344	10560

Table 4.4: Percentiles of the histograms [Hz].

As shown in Table 4.4, only 25% of the information is retained from the bird dataset when taking a sample rate of 8 kHz. For this sample rate, about 75% of the animal dataset frequency content is kept. However, we believe such information loss is unacceptable, requiring selecting a higher sampling rate. Since we want to minimize the loss of information, we decide to set a maximum loss of information below 10% for both datasets. At the 90th percentile, the sampling rate is upper-constrained by the bird dataset. Thus, we choose a sample rate of 16 kHz since it allows around 7% of frequency content loss for the animal dataset and about 9% for the bird dataset. Note that in most bird classification works, the authors use a significant sample rate, either 22.5 kHz or 44.1 kHz, because the frequency content is critical to distinguish bird species [144, 145]. However, in this work, we focus on an ultra-low-power device, and we therefore want to minimize the sample rate. Furthermore, we do not focus on the bird species classification but on a classification of events (birds, chainsaws, gunshots, and animals), allowing us to accept small losses of frequency content. With the sampling rate chosen, all the recordings are resampled to 16 kHz.

4.4.3 Resolution choice

Section 4.2.4 highlighted that the bit-depth of each dataset is the same except for the bird dataset, for which the resolution is unknown. To be consistent, we use the `soundfile` Python package to re-write all the bird recordings with a 16-bit resolution (in the format signed PCM 16 bits to be accurate).

4.4.4 Slice length choice

To train a convolutional neural network, we need the same size for the input images. Thus, we need the input audio recordings to have the same duration. To find the optimal audio input length (that is then processed into a fixed-size image), we need to use the information about the duration of each dataset, presented in Section 4.2.3. Taking a glimpse back at Figure 4.5, we observe that the choice is limited by the length of the gunshot and bird recordings since they are the shortest. The minimum length in the gunshot dataset is of exactly 1 second, while the minimum length in the bird dataset is of 300 milliseconds. Because it seems unfeasible to classify a sound based on a few hundred milliseconds, we choose a slice length of 1 second. If the choice of slice length was made bigger, we would lose a lot of gunshot recordings because they would not have the required length, which is not acceptable. It thus means that the network to be built will perform inference on 1-second slices. In reality, this is not a perfect length for bird classification. Many works, done in the context of the BirdClef challenges, for instance, use slices of length going from 3 seconds to 5 seconds to obtain a major part of the song or call of the bird [140, 146]. Note also that following the 1-second choice, 839 bird recordings are deleted from the dataset since they are smaller than 1 second in duration.

4.4.5 Segmentation of the recordings

To transform the audio samples into 1-second slices, two techniques are used:

- Because the chainsaw dataset always has a duration of 2 seconds (see Figure 4.5) and contains only relevant sound (see Section 4.2.6), we can make simple cuts of 1-second slices in each recording. To augment the number of samples for this class, we use an overlapping of 0.5 seconds. Therefore, each 2-seconds recording yields three slices (from 0 to 1 ; from 0.5 to 1.5, and from 1 to 2 seconds) ;
- The segmentation technique, proposed in Section 4.3, is used for the three remaining datasets. For each recording, we obtain a mask set to 1 when an important sound is detected and 0 otherwise. If we concatenate all these important portions of the mask and then cut them into 1-second slices, we obtain what we seek, but with a loss of temporal resolution. We must thus use another reasoning. We simply travel, in a deterministic order, over the whole recording, one second after the other, with an overlap of 0.5 seconds. We check if the analyzed 1-second slice contains at least $\mathbf{X}\%$ of the important signal from the mask in this specific cut of the recording. If it is the case, the slice is taken into the *Events* dataset. Otherwise, it is rejected. $\mathbf{X}$ is a threshold parameter found experimentally by trial and error. We want to select all slices containing a sound of interest without taking too much noise into account. A too-large value of $\mathbf{X}$ value does not take a lot of noise but misses important signal content. On the contrary, a too-small value of $\mathbf{X}$ threshold takes all important sounds but contains a lot more noise. It is thus again a trade-off that has to be made. Finally, $\mathbf{X}$ is chosen to be 15% for the animal and bird datasets and 35% for the gunshot dataset. The choice has been determined experimentally on around 20 samples from each class. We visually represent this process in Figure 4.14.

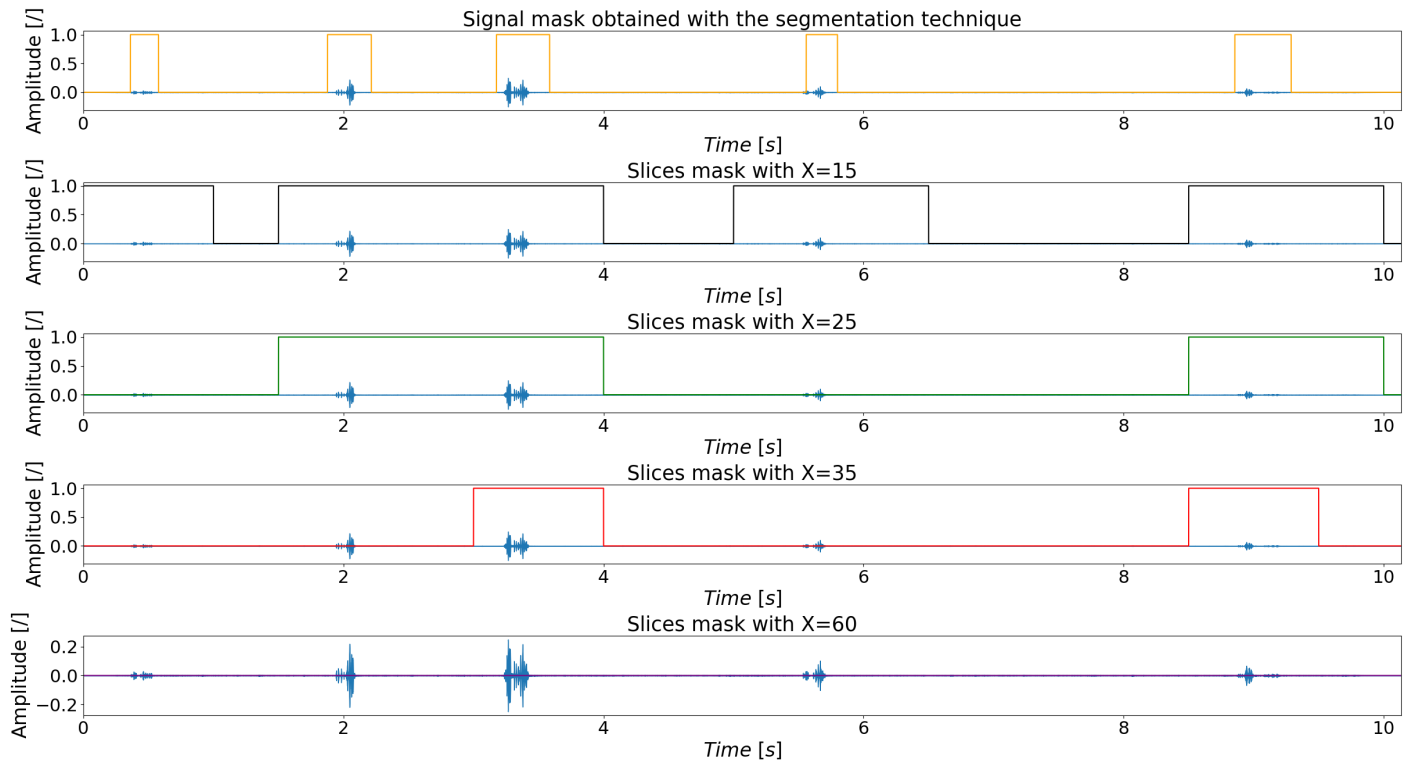


Figure 4.14: Bird sound segmentation. For each subplot, the blue waveform is the original sound. The first subplot represents the mask output obtained by the segmentation technique. The other subplots represent the mask for the 1-second slices with different $\mathbf{X}$ thresholds. In these masks, a value set to 1 means that this recording part is taken into the *Events* dataset.

As seen in Figure 4.14, a threshold value above 35% makes the recording pointless as no slice is kept for the *Events* dataset. The reason is that the important sounds are numerous but short in duration. Therefore, asking for large thresholds leads to long durations ($X = 35\%$ means that the important sound must be present at least 350 ms in the 1-second slice), suppressing all these short calls. Because this phenomenon of short sound is present in a vast majority of the recordings in our dataset, we must choose a small threshold value. The value of **15%** is selected for the animal and bird datasets since the sounds are often shorter than a gunshot sound with its echo. Note that we do not go below a value of 15% (for the animal and bird dataset) otherwise every glitch or error from the segmentation technique would yield a 1-second slice in the *Events* dataset.

4.5 Analysis of the *Events* dataset

The final dataset, ready to be used for training the convolutional neural network, contains 668 746 samples. Each audio sample lasts exactly 1 second, has a sample rate of 16 kHz and a resolution of 16 bits. The proportion of training samples in each class is shown in Figure 4.15. It demonstrates the large imbalance of this dataset.

Repartition of the number of 1-second audio slices among the 4 classes

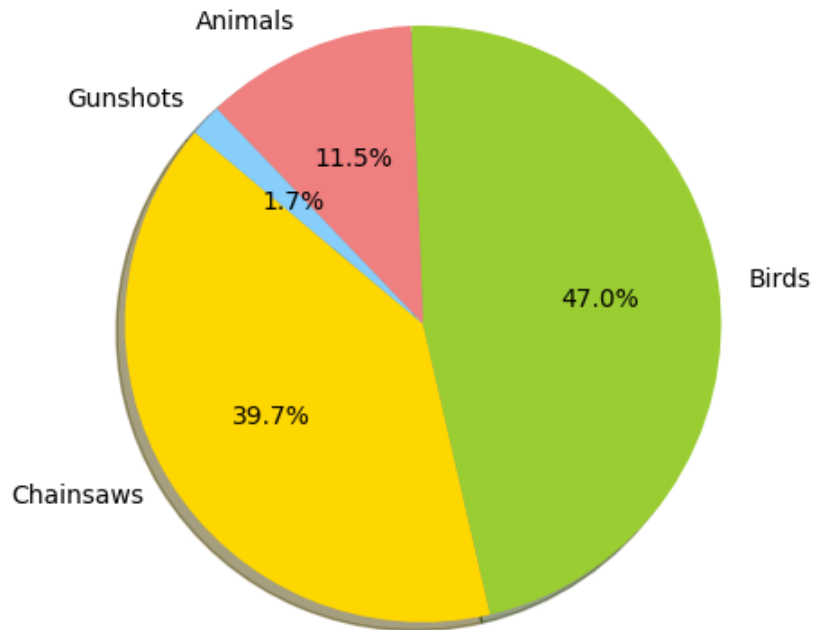


Figure 4.15: Proportion of the dataset among the four classes. The *Events* dataset is highly imbalanced with a large proportion of bird and chainsaw sounds.

4.6 Summary

In this chapter, we first explained the gathering of a relevant audio dataset fit for our classification task to distinguish between animal, bird, chainsaw, and gunshot sounds. With the help of *Rainforest Connection* and the *Borror Laboratory of Bioacoustics*, in addition to online databases known as *Xeno-Canto* and the *Gunshot Audio Forensics Dataset*, we were able to build a big imbalanced audio dataset composed of 101 387 recordings of animals, birds, chainsaws, and gunshots. By making an in-depth analysis of the compiled dataset, we discovered that the audio properties differed a lot depending on the class and the recordings. The sample rate, the number of audio channels, the recording duration, the resolution, and the audio quality are different for each class. Furthermore, except for the chainsaw recordings, the dataset contains a lot of irrelevant sounds. Therefore, there is a need to process this gathered dataset to train a convolutional neural network.

The changes made to the original dataset are:

1. **689** animal recordings were deleted from the dataset because these sounds represent species that are known to live outside of forest environments ;
2. The sample rate of each recording was resampled to **16 kHz** since this value has been found to be optimal for our dataset. The result comes from a frequency analysis for which the aim is to find the lowest sample rate that limits, in mean added by the standard deviation, the loss of frequency content to less than 10% ;
3. The resolution of the bird recordings has been modified to fit the format **signed PCM 16 bits** because the initial bit-depth for this class was unknown ;
4. The recordings have been segmented to create **1-second** samples to be used as input of a convolutional neural network. This specific choice comes from the shortest length of the dataset. If we chose a bigger length, some recordings would have been unusable. To create 1-second slices, we used three cuts for the chainsaw recordings, happening from 0 to 1 second, 0.5 to 1.5 seconds, and 1 to 2 seconds. For the remaining classes, we used a segmentation technique inspired from [140] to output a mask containing, for the whole recording, the parts containing relevant sound and the noise parts. Then, we traveled over the recording and took each 1-second analyzed slice if at least 150 ms of important sound is present in it for the animal and bird sounds and at least 350 ms for the gunshot sounds.

Finally, the modified dataset called the **Events** dataset is analyzed. It contains in total **668 746** samples. From this, **47%** of the samples are bird recordings, **39.7%** are chainsaw audios, **11.5%** belong to the animal dataset, and **1.7%** to the gunshot dataset. There is thus a large imbalance in the different categories to classify.

CHAPTER 5

TRAINING OF A BASELINE CONVOLUTIONAL NEURAL NETWORK ON THE *EVENTS* DATASET

Because the classification task we target has never been done before, we cannot compare our results with the state-of-the-art. Thus, we choose a *MobileNetV1* network [13] as a baseline model since it is a powerful architecture that has been thought and studied for embedded devices. To obtain the most optimal model in terms of energy efficiency and accuracy, we perform a set of experiments optimized for the task. To that purpose, we first introduce the setup used for the training of the network as well as the baseline model architecture. Then, we present each experiment allowing us to obtain the most optimal baseline model.

5.1 Training setup

The setup to run all the experiments consists of a computer with an *AMD Ryzen 7 3700x* processor and an *NVIDIA Geforce RTX 2070 Super* (8GB of RAM) GPU. The computer is also equipped with 16 GB of RAM and an *SDSSDH3-2T00 Solid-State Drive* (SSD) of 2 TB to speed up reading and writing audio and images. This setup is chosen to accelerate the training process while remaining affordable. The experiments presented in the remainder of this chapter can, of course, be performed on any other low-cost machine that supports *Tensorflow* and *Keras*. The training time will just be longer.

In this entire chapter, except if it is clearly specified, we consider, for each experiment, the hyperparameters presented in Table 5.1.

Optimizer	SGD
Learning rate	0.001
Batch size	128
Maximal number of epochs	200
<i>Keras</i> callbacks	learning rate scheduler with step_decay every 10 epochs with a factor of 0.5 ; early stopping on the validation loss with a patience of 15
Loss	weighted categorical cross-entropy

Table 5.1: General hyperparameters used for the experiments.

Note in Table 5.1 that a callback in a given framework is an object able to perform actions at various stages of the training. The *learning rate scheduler* callback allows modifying the learning rate during training after a specific number of epochs. *Step_decay* is a schedule that is passed as an argument to the *learning rate scheduler* callback. The Python code, as well as an illustration of the schedule, is found in Appendix A.2. This schedule allows dropping the learning rate by a given factor every few epochs. Other *Keras* schedules are possible and can be found in [147]. The *Early Stopping* callback allows stopping the training process when a monitored metric has not improved for a given number of epochs. The metric to monitor as well as the number of epochs are arguments of the callback. More information about it can be found in [148], while all *Keras* callbacks can be found in [149].

It is not possible due to the large number of experiments, the big dataset, and the limited computational power of the used machine to perform a hyperparameter search over the three first hyperparameters of Table 5.1 for each experiment. If we had an infinite amount of computing resources, the best idea would be to do all the experiments presented in Section 5.3 in one shot directly with the optimal hyperparameters. In practice, this would take a considerable and unmanageable amount of time and this is why the flow of experiments explained in Section 5.3 is proposed, based on [150].

Note our choice for the initial hyperparameters :

- The SGD optimizer is chosen because it is known to be one of the most used in computer vision tasks [151, 152] ;
- The learning rate is taken not to be too big or too small (since it decreases when the *learning rate scheduler* callback is performed). A value of 0.001 is a good compromise, and a known value in machine learning [153] ;
- In a lot of works, batch size values in {16, 32, 64, 128} are often chosen [140, 2]. We decide to choose the biggest value (128) from this range to speed up the training ;
- The maximal number of epochs is not the most important hyperparameter since the *early stopping* callback should stop the training in the first place. We choose a value equal to 200 to give the network enough epochs to learn ;
- A weighted cross-entropy loss is chosen. This is not a common loss in *Keras*, and it has therefore been coded thanks to [154]. The principle is to give a larger weight to the classes with a smaller number of training samples. It is useful for imbalanced datasets. Its implementation in Python within the *Keras* framework is shown in Appendix A.3.

Finally, here is an important note about the way the features are processed before being given as input to the convolutional neural network. Before the training, each feature is computed, rescaled, and cast to eight bits and then written on the SDD. During training, each feature is normalized pixelwise on-the-fly by removing the mean of the image and dividing the result by the standard deviation of the feature.

5.2 Baseline model

Usually, the baseline model is chosen as the network of another work in the literature solving the same task to be able to compare the results. However, the task we solve has never been done in the literature. Thus, the baseline model is chosen in another way. To quantify the maximal performance we can get on this task, we choose the *MobileNetV1* architecture, a powerful Convolutional Neural Network specialized for embedded machine learning [13]. It has been used in many works dealing with

embedded devices such as [155, 156, 157, 158]. As explained in [13], the network uses, except for the first layer, depthwise separable convolutions, which are the core layers of this architecture. Concretely, using this specific layer instead of the normal convolution allows for a reduction in computation by a factor $\frac{1}{N} + \frac{1}{D_K^2}$, where N is the number of output channels (feature maps), and D_K is the height-width of the convolution filters, at the cost of a small drop in accuracy on benchmark datasets. *MobileNetV1* reaches 70.6% accuracy on *ImageNet* while VGG-16 obtains 71.5% at the expense of $27\times$ more MACC operations and $33\times$ more model parameters. On the *Stanford Dogs* dataset, the original version of *MobileNetV1* achieves 83.3% while InceptionV3 obtains 0.7% more for 5000 million MACC operations and 23.2 million parameters compared to 569 million MACC operations and 3.3 million parameters for *MobileNetV1*. Other results are demonstrated in [13]. The complete architecture of *MobileNetV1* is shown in Figure 5.1. It is worth mentioning that the authors of *MobileNetV1* introduced an architecture hyperparameter α called the width multiplier. It allows obtaining a smaller architecture for highly constrained applications in terms of computing power. This parameter lies in the range $[0 ; 1]$ and is equal to 1 initially. Playing on this parameter allows modifying the number of input channels and the number of output channels. More specifically, the number of input channels M becomes αM while the number of output channels N becomes αN . Therefore, the cost of a depthwise separable convolution is further reduced, knowing that it considers the number of input and output channels in its computation. Table 5.2 shows the impact of the width multiplier on the accuracy as well as on the number of operations and parameters of the model. Finally, the input shape for the original version of this architecture is a square image of 224×224 pixels. However, this shape can be modified according to the application requirements.

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

Figure 5.1: Architecture of the *MobileNetV1* network [13]. Except for the first layer which is a normal convolution with a 3×3 kernel, **Conv** refers to a pointwise convolution layer followed by a batch normalization layer and a ReLU activation. **Conv dw** refers to a depthwise convolution layer followed by a batch normalization layer and a ReLU activation.

Width multiplier	<i>ImageNet</i> accuracy	Million Mult-Adds	Million parameters
1.0 MobileNet-224	70.6%	569	4.2
0.75 MobileNet-224	68.4%	325	2.6
0.5 MobileNet-224	63.7%	149	1.3
0.25 MobileNet-224	50.6%	41	0.5

Table 5.2: *MobileNetV1* width multiplier impact [13]. Reducing the width multiplier allows considerably decreasing the number of MACC operations as well as the number of parameters of the model. However, it comes at the cost of a loss of accuracy.

MobileNetV1 fame and efficiency gave rise to the second version of *MobileNet* networks known as *MobileNetV2*. This family of networks is also actively used in computer vision tasks [159].

5.3 Experiments flow

All the experiments performed with the baseline model are represented in Figure 5.2. Each following subsection presents one experiment. Note that the entire dataset is only used for the last experiment. Using the complete dataset for all experiments is impossible in terms of computing power. To perform the training on the last experiment, the *Events* dataset has been divided into the following proportions : 80% of training set, 10% of validation set and 10% of test set. Also note that the *Events* dataset is highly imbalanced with a small number of gunshots and a high number of birds for example. This imbalance has been kept in all sets. The used dataset for experiments 2, 3, and 4 depends on the result of the first experiment. For each result, if it is not clearly specified, the validation accuracy is obtained by taking the mean of this metric over the last five epochs. For the computation of the confusion matrix, we use the trained model up to the last epoch. Note that the purpose of this flow of experiments, based on [150], is to find the most optimal feature set, data augmentation technique, and hyperparameters from the reduced dataset and directly reuse those on the complete dataset.

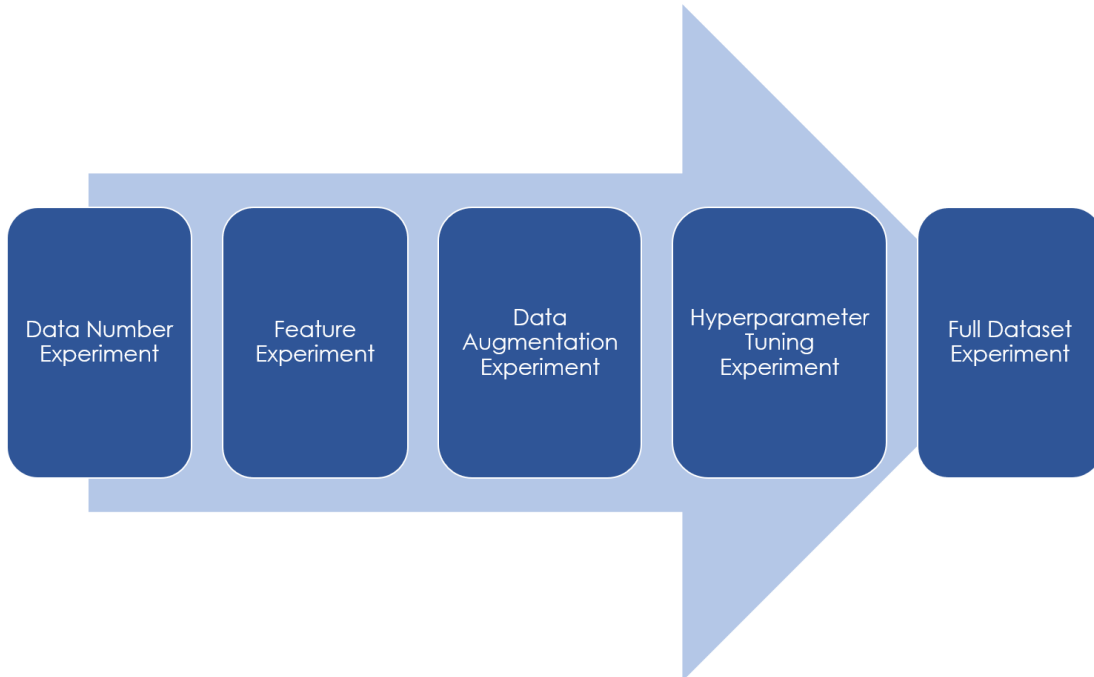


Figure 5.2: Diagram flow for all the experiments done with the baseline model.

5.3.1 Impact of the number of training samples on the classification

The objective of this experiment is double : quantify, in a naive way, the performance that *MobileNetV1* is able to give for the classification task as well as see the impact of the number of training samples on the validation accuracy. For this specific experiment, we train the model with 5%, 10%, 50%, and 99% of the entire *Events* dataset. Note that we floor the number of training samples if the proportion gives a float number. Each augmentation of the training dataset contains in it the previous one as well as new training data. For instance, the 50% training dataset includes the 10% dataset and 40% of new training data. Also note that each training set has the same imbalance as the original one so that the *weighted_categorical_crossentropy* loss still has a meaning in this experiment. The model is tested on a validation dataset which is the same for every training dataset and equal to the remaining 1% of the *Events* dataset. Taking the same validation set for all training sets allows us to compare their effects on the same scale. Finally, the feature (and its hyperparameters) used to feed the *MobileNetV1* network is obtained randomly among the set of features presented in Section 2.6.3 : the Mel-spectrogram feature with a Hanning window of 1024 samples, 25% overlap, and 32 mel filters is the chosen one. We plot the result in Figure 5.3.

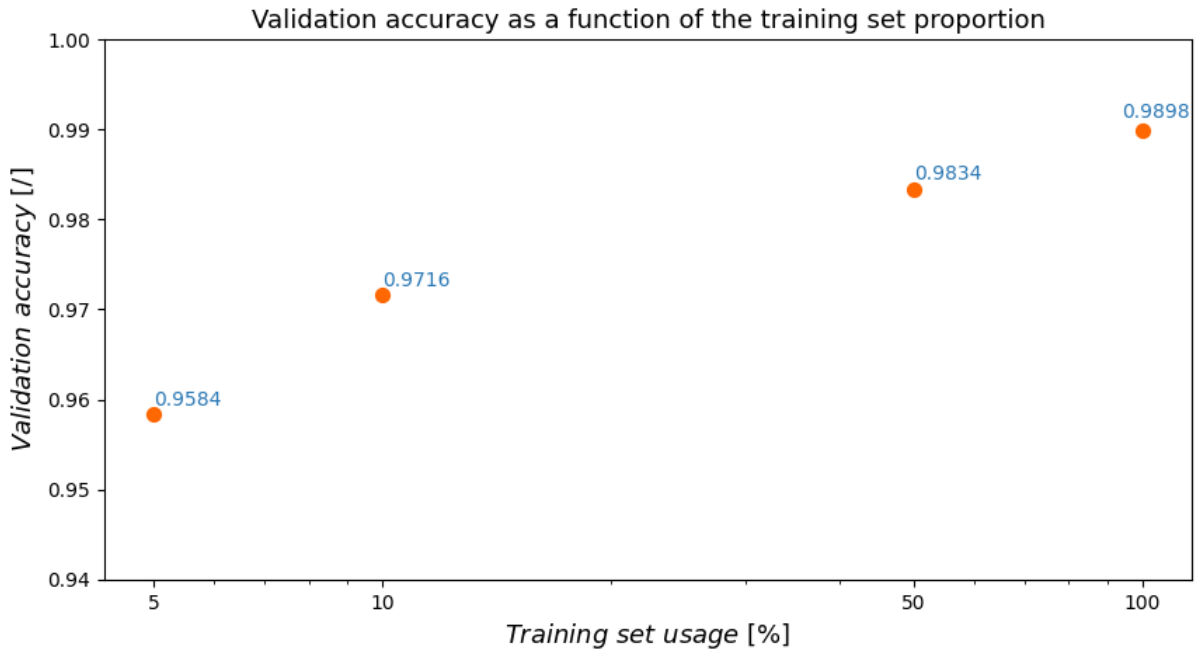


Figure 5.3: This plot shows the positive impact of the number of training samples on the validation accuracy.

Two conclusions can be drawn :

- The validation accuracy is high (bigger than 95%). One possible reason could be that the CNN architecture can learn important and distinguishable features among the four classes. Another reason could be linked to the quality of the original dataset as well as the sequence of pre-processing operations followed to obtain the *Events* dataset. We can also imagine that the distinction between birds and animals classes is easier than what was initially thought ;
- Adding more training data for the classification task has a positive impact on the validation accuracy, which comforts us in the choice we made to obtain a lot of training data.

Because of this result, we decide to use 5% of the *Events* dataset for the remaining experiments,

except the last one (**Full Dataset Experiment**). The proportions remain the same as before with 80% for the training set and 10% for the validation and test sets. The imbalance between classes is kept in each set. We choose 5% of the *Events* dataset because the accuracy is already high with this small set and it allows to speed up the training process since fewer data are available.

5.3.2 Choice of the feature type

In this experiment, the seven features presented in Section 2.6.3 are studied. The purpose is to find the best combination of feature type and feature hyperparameters for the classification task while requiring the lowest resources for the feature processing block and the network inference. We focus our study on three aspects : the *complexity*, the *number of pixels in the input slice*, and the *validation accuracy*. 178 variations of the seven features are studied. The complete list of all these variations, with their specific hyperparameters and their chosen name, is shown in Appendix A.4 from Table A.1 to Table A.7. The variations are chosen because they are known and used in the literature. Note that the type of window used to perform the short-time Fourier Transform is not part of the studied hyperparameters. We choose a Hanning window because "in general, the Hanning window is satisfactory in 95% of cases. It has good frequency resolution and reduced spectral leakage" [160].

5.3.2.1 Complexity

The first aspect studies the complexity of processing 1-second audio into the desired feature. Because we target an ultra-low-power device, the feature processing step must be taken into account since it plays a major role on the power consumption. Figure 5.4 represents the time (on a logarithmic scale) required to compute each variation of each feature on the setup machine. The machine is of course more resourceful than an embedded device, but the time-consuming features remain the most difficult to process regardless of the hardware.

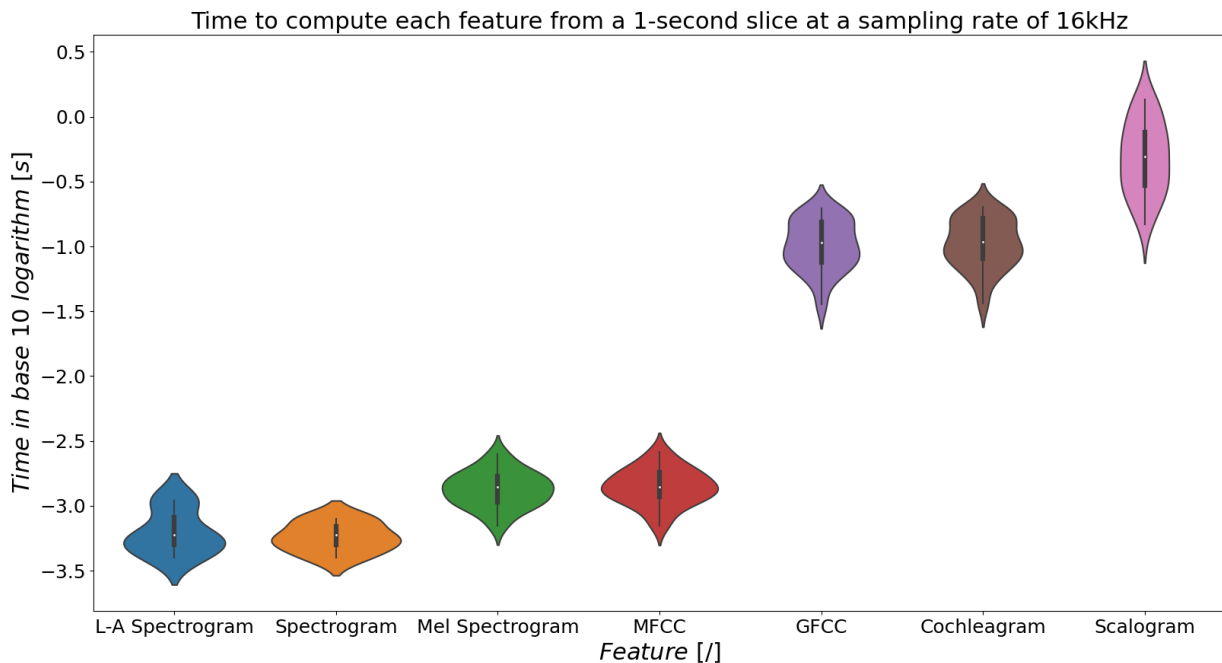


Figure 5.4: This figure is a violin plot and represents the time required to compute each variation of the seven studied features on a logarithmic scale. A bigger width involves more variations of the feature in the same y-coordinate. **L-A Spectrogram** means logarithmic-amplitude spectrogram. We note that the four features on the left are the best candidates as they need far less time to be computed.

As it can be seen in Figure 5.4, three features (*GFCC*, *Cochleagram*, and *Scalogram*) are well above the four other ones in terms of time needed to compute the feature on a logarithmic scale. These are thus not viable options for an ultra-low-power embedded device. The *GFCC*, *Cochleagram*, and *Scalogram* features are thus discarded from the best possible feature set. Only 70 variations remain among the four kept features.

5.3.2.2 Number of pixels in the input slice

A convolutional neural network is highly dependent on the input feature in terms of MACC operations. Because the input feature is an image, we can talk about the number of pixels in the input slice. The bigger this number, the larger the number of MACC operations, and the stronger the consumed energy of the network. As a proof of concept, the following table shows the number of MACC operations of the *MobileNetV1* architecture with different input shapes.

Input Shape	(224, 224, 3)	(32, 94, 1)	(32, 59, 1)
MACC operations	572 821 068	29 457 484	17 613 388

Table 5.3: The number of MACC operations of the model decreases with the number of pixels in the input image. The values are obtained with the *keras-flops* Python package.

This study attempts to obtain an input slice as small as possible to limit the energy consumed by the network for an inference. Figure 5.5 thus shows the number of pixels of each variation of each remaining feature in a violin plot. We can notice that the *spectrogram* and *logarithmic-amplitude spectrogram* features have in the majority an input slice bigger than the ones from the *mel spectrogram* and *MFCC* features. The explanation is simple and comes from the way each feature is computed. Each feature has the same number of temporal frames and thus the same number of columns in the image. However, the two features on the left of the figure have a number of rows equal to $\frac{FFTsize}{2} + 1$, while the *mel spectrogram* has a number of rows equal to the number of mel filters and the *MFCC* feature has a number of rows equal to the number of mel-frequency cepstral coefficients. Because the FFT size is largely superior compared to the two other parameters, it is logical that the *spectrogram* and *logarithmic-amplitude spectrogram* features require a larger number of pixels in the image. We thus keep the two features on the right of the plot leading to 48 remaining variations. Nevertheless, as a good practice, we also keep one variation from the *logarithmic-amplitude spectrogram* feature set to compare the final validation accuracy.

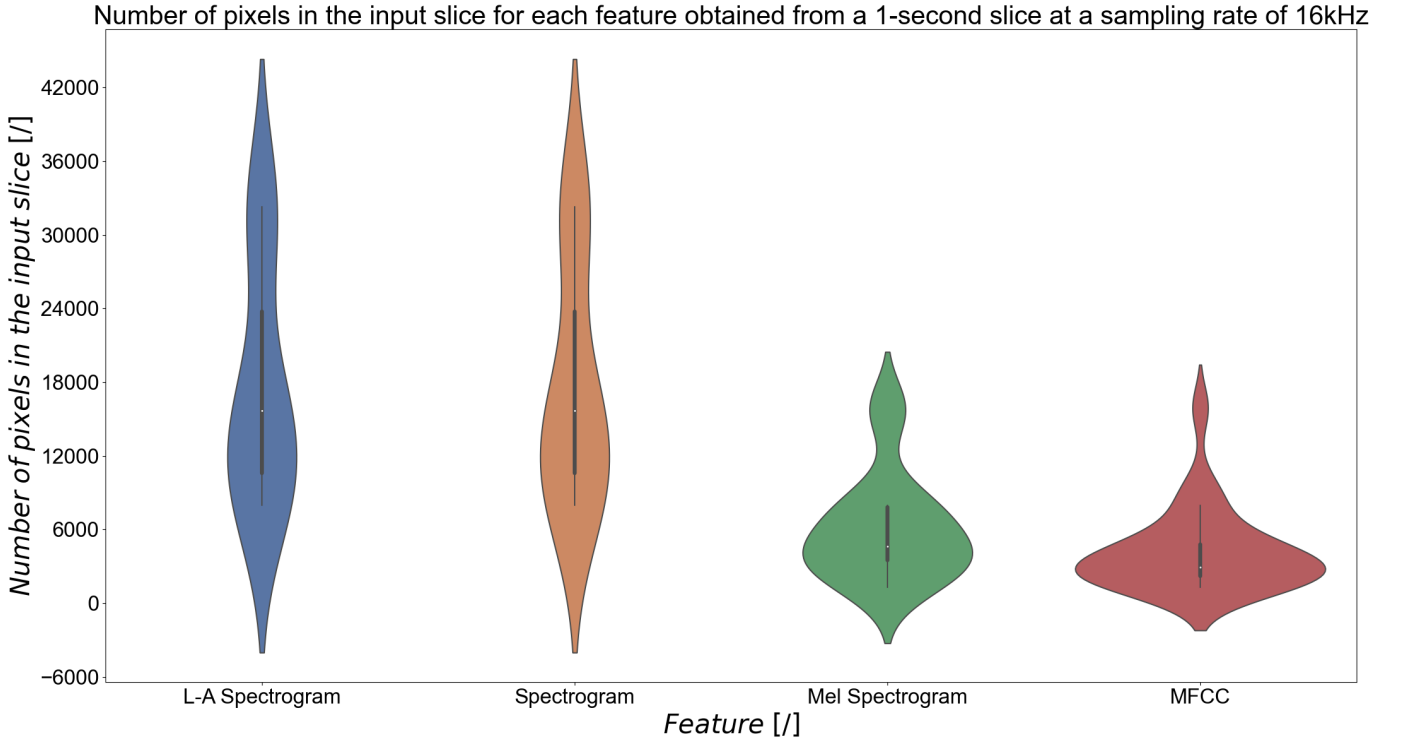


Figure 5.5: Number of pixels in the input slice for each feature variation. **L-A Spectrogram** means logarithmic-amplitude spectrogram. We note that the two features on the right are the best candidates as they involve far fewer MACC operations due to their smaller shape.

5.3.2.3 Validation accuracy

By training the *MobileNetV1* architecture with the 49 different feature sets, we obtain 49 validation accuracies. We plot in Figure 5.6 the validation accuracy as a function of the number of MACC operations needed by the model (since it depends on the number of pixels of the feature). The Pareto-optimal curve is also represented on the plot allowing us to find a good trade-off between the lowest number of MACC operations and the highest validation accuracy. We finally choose the top-left feature, named the MFCC-20 variation in our experiments. It represents the MFCC feature with a Hanning window of 512 samples, 75% overlap, 128 mel filters, and 39 mel filter cepstral coefficients. This feature set has almost the smallest number of MACC operations for only a small drop in terms of validation accuracy.

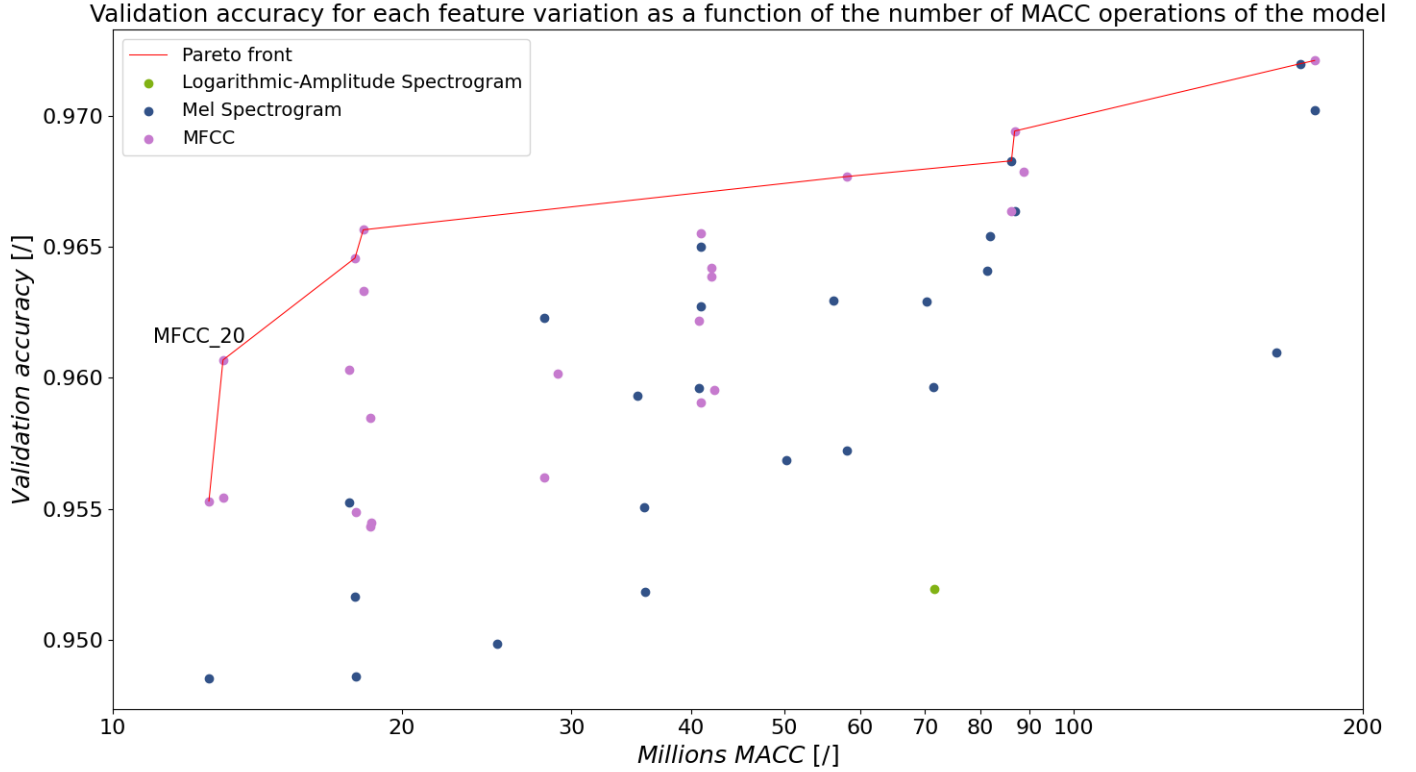


Figure 5.6: Validation accuracy as a function of the number of MACC operations of the network. The Pareto front is also represented and its implementation is based on [161].

5.3.3 Choice of the data augmentation technique

For this experiment, the data augmentation techniques that are studied are the ones presented in Section 2.5.1.2. We perform multiple schemes to find the set that allows giving the best boost in validation accuracy. Among the different schemes, we try all the raw augmentation techniques alone, all the raw techniques together, the spectral augmentation alone, and all the techniques together. For each augmentation technique, the original samples are kept, and the dataset thus always grows in size compared to the original one. In reality, a lot of augmentation techniques exist and trying all of them could be beneficial. However, as usual, it is not feasible in terms of computing power and time we have at our disposal. This is why this subset of augmentation techniques, being the most known and most efficient, have been chosen as stated in Section 2.5.1.2.

The parameter(s) of each audio augmentation technique are, amongst others, based on [162] which studies in detail audio data augmentation for animal sound classification :

- *Time Shift* : we perform it randomly inside the entire recording towards the left or the right. Because we work with 1-second audio at a sampling rate of 16 kHz, the time shift is chosen randomly in the interval $]-16000 ; 16000[$;
- *Pitch Shift* : we increase the pitch by two semitones and decreases it by two semitones as well. This technique gives two new audio samples per recording with this setting ;
- *Time Stretch* : we increase the speed of the signal by 15%, zero-padding the end of the recording to fit the 1-second slice ;
- *Noise Injection* : we do not add gaussian noise with a given standard deviation. We add

environmental noise coming from the field and obtained by *Rainforest Connection* as mentioned in Table 4.1. For each recording, we add on top of the audio 40% of this environmental noise ;

- *Mix-Up* : the α factor in Equations 2.7 and 2.8 is sampled from a Beta probability density function for which the two parameters of the distribution are chosen to be equal to 0.4 ;
- *Frequency & Time masks* : three indices are hidden (set to zero) along the width and height axis. Thus, the two masks (frequency and time) have a width height of 3 columns-rows.

It should be mentioned that there exist some Python frameworks that allow doing audio data augmentation such as *audiomentations* [163] or *pydiogment* [164]. However, it is not used in this work because some of the raw techniques, such as the *noise injection* augmentation, were performed differently or not available (even though they are now). We also decided not to perform data augmentation on-the-fly but before the training process, thus increasing the dataset size. The reason is that modifying the *Keras* implementation to handle all of our augmentation schemes required more work. In practice, audio data augmentation techniques can be performed on the raw data or the spectral feature. It has been chosen to do it, when possible, always directly on the audio waveform.

Finally, the confusion matrix before this experiment (after choosing the best feature set obtained in Section 5.3.2) is computed and shown in Figure 5.7. By analyzing it, we notice that the chainsaws class is already extremely well-classified, and we thus decide not to augment it. The meaning of it is that *Time Shift*, *Pitch Shift*, *Time Stretch* are not done on chainsaw waveforms. *Mix-Up* is performed, but the α coefficient is changed accordingly for chainsaw samples such that the new virtual sample never appears to be a new chainsaw sample. However, *Noise Injection* is performed since we want to be more robust to the forest field, and *Frequency & Time masks* is also applied.

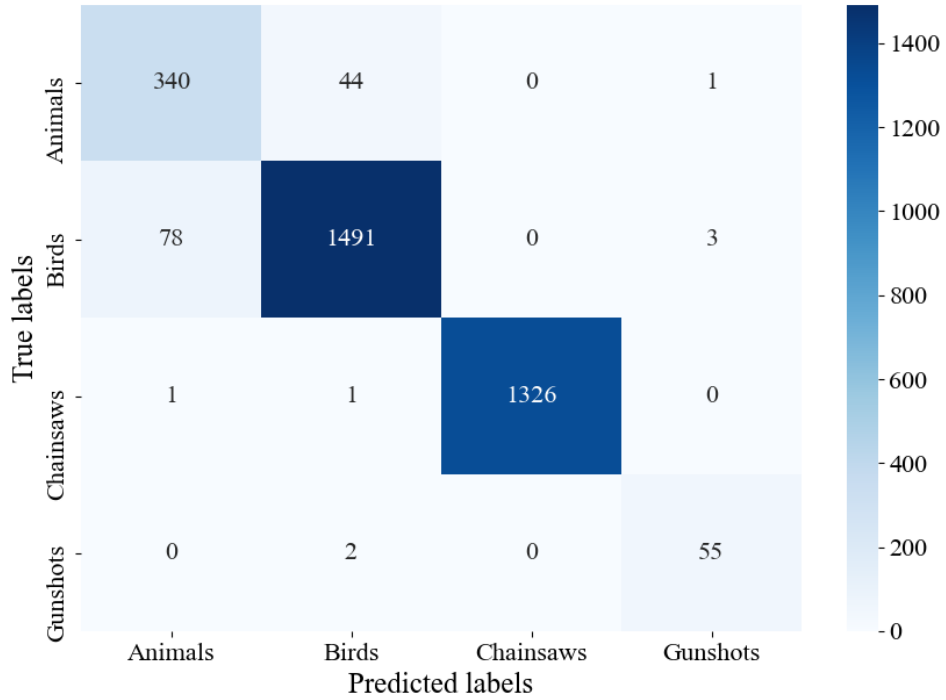


Figure 5.7: Confusion matrix on the validation accuracy of the baseline model before the data augmentation experiment, after the selection of the optimal feature type (MFCC_20).

The result of the experiment is shown in Figure 5.8. As mentioned in Section 5.3, the validation accuracy is obtained by taking the mean over the five last epochs. However, we perform in this case three times the experiment and we take the mean of the 3 obtained validation accuracies.

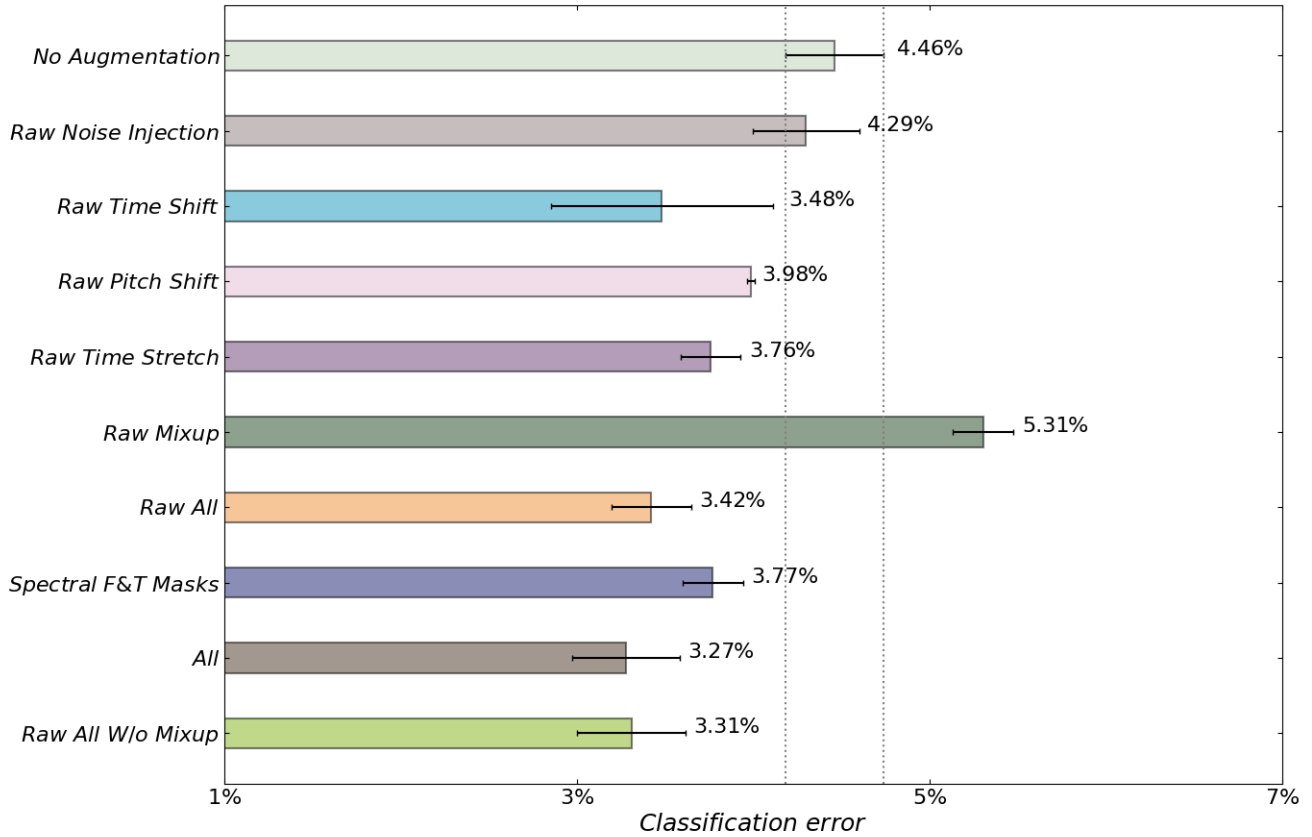


Figure 5.8: Classification error on the validation set for different augmentation methods. Bars and whiskers indicate the mean and its 95% confidence interval computed from three repetitions of each experiment.

Some conclusions can be drawn :

- Against all odds, the *Mix-Up* augmentation technique, which is known to be efficient in audio classification tasks [47, 48], performs poorly on this dataset. It increases the classification error by approximately 1% ;
- Our results are consistent with [165] and [166], which focus respectively on *Singing Voice Detection* and *Musical Instrument Recognition*. *Noise injection* does not significantly improve the classification as in [166], while it decreases the accuracy in [165]. *Pitch Shift* and *Time Stretch* improve the validation accuracy almost similarly as in [166], while the *Pitch Shift* improvement is better in [165], allowing a 25% classification error reduction compared to the original case. The *Mix-Up* effect is negative on the accuracy as it is the case in [165] ;
- *Time Shift* has a good improvement on the classification error, but its 95% confidence interval is larger than with the other augmentation techniques challenging the hypothesis of best augmentation technique. Combining all of our raw augmentation techniques, such as in [162], allows further reducing the classification error ;
- The spectral feature *Frequency & Time masks* also improves the accuracy in the same range as the best raw audio techniques alone. The *All* row in Figure 5.8 considers all the techniques and

the accuracy is improved slightly compared to only the raw audio techniques. Finally, the last row *Raw All W/o Mixup* is a combination of all the raw audio augmentation techniques except the *Mix-Up* since the latter harms the accuracy. We see that it almost matches the *All* row with a loss of 0.04% in mean accuracy. This last presented technique is finally chosen because it gives to within 0.04% (statistical noise) the best result and the number of training samples is equal to half of the number of samples that would be obtained with all the techniques taken together. Choosing all the augmentation techniques in both domains (temporal and spectral) would double the training time, which is already a limiting factor.

The difference in the confusion matrix before and after the data augmentation experiment is shown in Figure 5.9. Even if the validation accuracy has not improved a lot, the animals class has obtained a big boost (5.5%) in classification accuracy. The reason is that the validation set is split taking into account the imbalance of the classes. Because the animals dataset is only 11.5% of the total used dataset, the validation accuracy does not improve significantly. The birds class has improved by 1.1%, while the chainsaws class has been negatively impacted by data augmentation by 0.8%. The reason comes from the fact that it has been decided not to augment chainsaw samples because the accuracy of this class was already really high. Finally, the gunshots class has not been affected. The objective of the difference of confusion matrix is to see improvements in the diagonal of the matrix and reductions in the non-diagonal elements. We thus want to improve the true positives of each class and reduce the false positives.

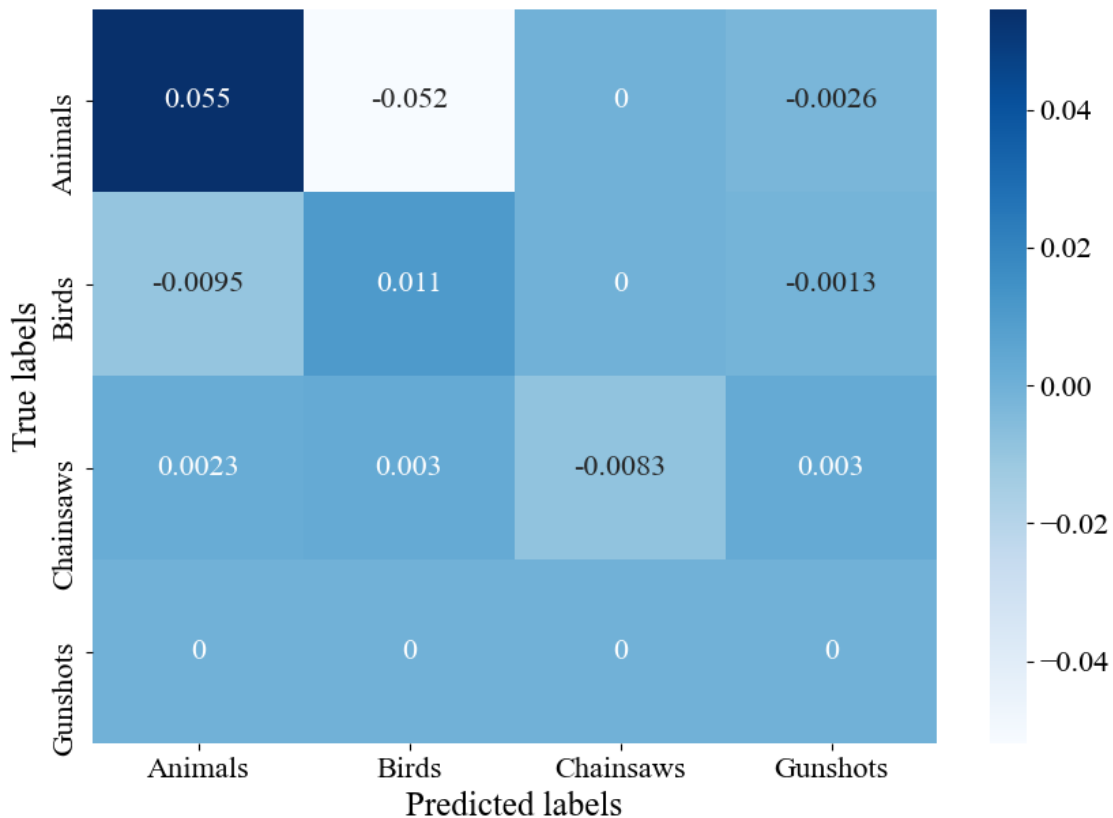


Figure 5.9: Difference of confusion matrices with and without data augmentation. Positive values indicate a positive change in confusion and negative values indicate a negative change in confusion.

5.3.4 Tuning of the training hyperparameters

As mentioned in Section 2.2, the training hyperparameters have a critical influence on the convergence of a neural network. Therefore, it is paramount to choose them wisely. This step is called *hyperparameter tuning*, as we search for the best possible hyperparameters with our model and dataset.

For the hyperparameters tuning, a lot of libraries exist, each one supporting its own algorithms and having its proper advantages and disadvantages. Ten of the most known are reported in [167], which include *Ray-Tune* [168], *Talos* [169], *Optuna* [170], and others. In this work, we have decided to choose the latter as our tuning framework since a lot of recent articles see it as the best hyperparameter tuning framework to date [171, 172]. *Optuna* [173] is easy to implement, contains multiple optimization algorithms, and has different visualization tools at its disposal. In terms of algorithms, the best idea would be to make a grid search¹ among a specific search space. However, because the model is complex and the dataset big, even with only 5% of the whole training set, it is not possible to allow this algorithm in terms of computing power. Moreover, a random search² is not a viable option since all the trials are taken at random and a good trial could be missed easily. The second best algorithm would thus be based on a Bayesian optimization. Luckily, *Optuna* proposes a TPESampler³, known as the *Tree-Structured Parzen Estimator* algorithm, which is close, in practice, to a Bayesian optimization. The principle is explained in [174].

The architecture of the baseline model being fixed, the only hyperparameters that can be tuned are the ones from the training process : the learning rate, the batch size, and the optimizer. The search space that has been chosen is the following :

```
1 batch_size = trial.suggest_categorical('batch_size', [16, 32, 64, 96, 128])
2 optimizer_name = trial.suggest_categorical("optimizer", ["Adam", "SGD"])
3 lr = trial.suggest_categorical("l_r", [5e-5, 1e-4, 5e-4, 1e-3, 5e-3, 1e-2])
```

Concretely, batch sizes in the set {16, 32, 64, 96, 128} are in the search space for the reason explained in Section 5.1. In terms of optimizers, the *SGD* optimizer is used as the baseline one, and we also add the *Adam* optimizer, which is an adaptive method also known in the literature for its performance [152]. For the learning rate, we try several values since the learning rate might be, based on experiments, the factor that influences the most the training process [175].

We perform the hyperparameter tuning with the *Tree-Structured Parzen Estimator* algorithm from *Optuna* and 30 trials. This specific value has been chosen as half of the entire search space. The Bayesian algorithms work by sampling first at random the first trials. The TPESampler does the same thing, and we set 10 random trials first. The mean accuracy over the five last epochs is still retrieved as the metric of interest. The final validation accuracy result is given in the following table as well as the one obtained after the data augmentation experiment. We obtain almost a 2% improvement in classification accuracy.

¹Grid search means testing all the possibilities (states) from the search space.

²Random search means testing randomly a given number of possibilities (states) from the search space.

³A sampler in the *Optuna* framework is a class that allows to sample parameters of a search space according to a given algorithm (given in the name of the sampler).

Before hyperparameter tuning	96.67%
After hyperparameter tuning	98.58%

Table 5.4: Validation accuracy of the baseline model before and after hyperparameter tuning

A slice plot coming from the *Optuna* framework tool is shown in Figure 5.10. It allows seeing how the search space has been studied and which part, more specifically, has been focused on by the Bayesian optimization.

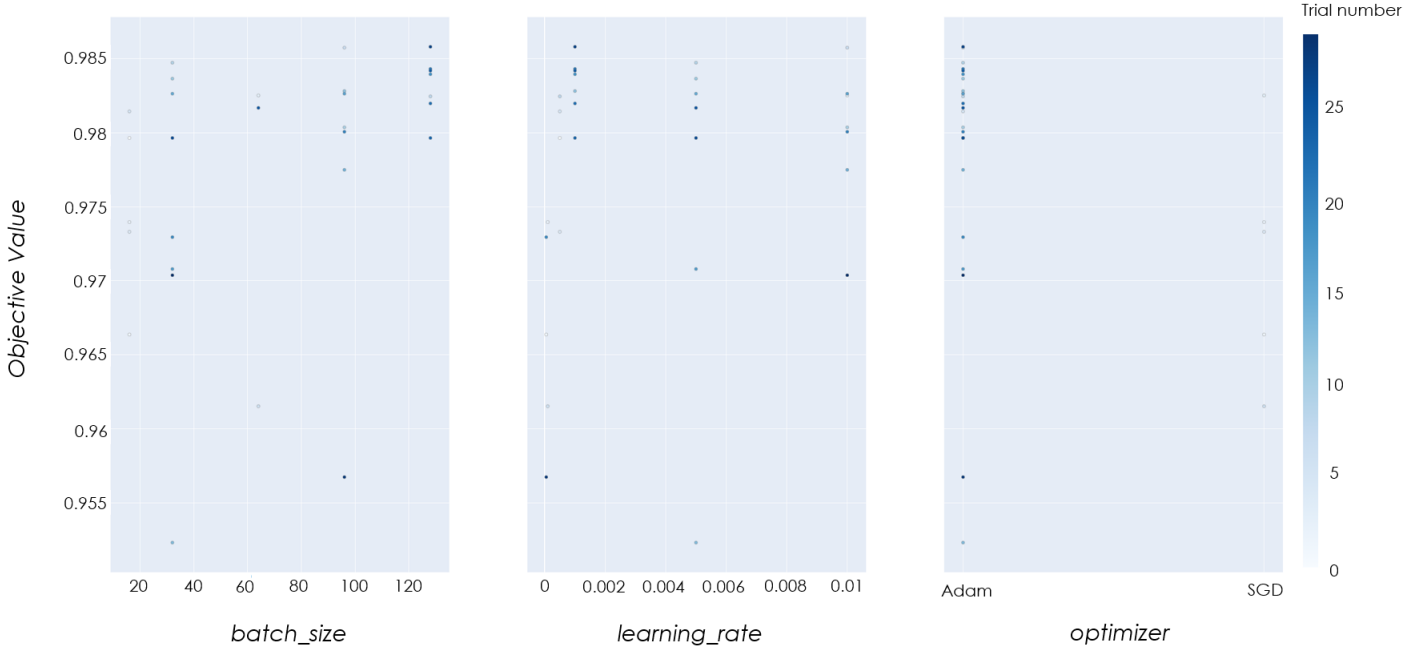


Figure 5.10: Slice plot of the hyperparameters search space.

We notice that the *Adam* optimizer has been the focus of the search compared to the *SGD* optimizer. In terms of batch size, the search has focused on the low values (*16,32*) and the high values (*96,128*), but only three trials were done for the value *64*. All learning rates have been tested. Finally, the best result in terms of validation accuracy is obtained, with an *Adam* optimizer, a batch size of *128*, and a learning rate of *1e-3*. Initially, the *SGD* optimizer, a batch size of *128*, and a learning rate of *1e-3* were used. Therefore, the optimizer change is responsible for the 2% classification improvement.

5.3.5 Optimized baseline model

When we have done all the proposed experiments on the reduced dataset to find the feature type, the data augmentation techniques, and the tuned hyperparameters, it is time to transfer this knowledge to the complete dataset. Therefore, we train the baseline model with the entire dataset. The data augmentation technique that has been chosen (*Raw all W/o Mixup*) transforms the original audio dataset of 668 746 training samples into a set of 2 359 816 samples. We then preprocess all these audios into MFCC features computed with the following hyperparameters : a Hanning window of 512 samples, 75% overlap, 128 mel filters, and 39 mel filter cepstral coefficients. The training setup changes slightly compared to the other experiments : it uses the *reduceLRonplateau* callback instead of the *learning rate scheduler*. This callback allows reducing the learning rate by a given factor only when a monitored metric fails to improve after a specific number of epochs. We choose to monitor

the validation loss and to reduce the learning rate by half after seven epochs if the validation loss has not improved. The training hyperparameters are obtained at the output of the previous experiment (*Adam* optimizer, batch size of *128* and learning rate of $1e-3$).

The training lasts for 51 epochs and takes 16 minutes per epoch. The accuracy and loss plots as a function of the number of epochs are shown in Figure 5.11. The validation curves are not as smooth as the training curves. The little instabilities at the beginning of the training are probably due to the higher learning rate. In the beginning, the learning rate value is high (0.001), and we are far from convex optimization, and it decreases during the training, creating a more convex behavior and stabilizing the curves. Therefore, all smooth processes related to convex optimization appear starting from the 30th epoch approximately. Concretely, using a high learning rate can lead to big changes in the training process that can be explained by an exit of a local minimum. Also note that the validation accuracy is higher than the training accuracy due to the dropout layer in the fully connected layer. The behavior of the dropout layer is different during training and evaluation. When training, a percentage of the hidden units is set to 0 (0.1% in our case). When testing, all hidden units are used. Therefore, the model during evaluation is more robust and can lead to a higher validation accuracy.

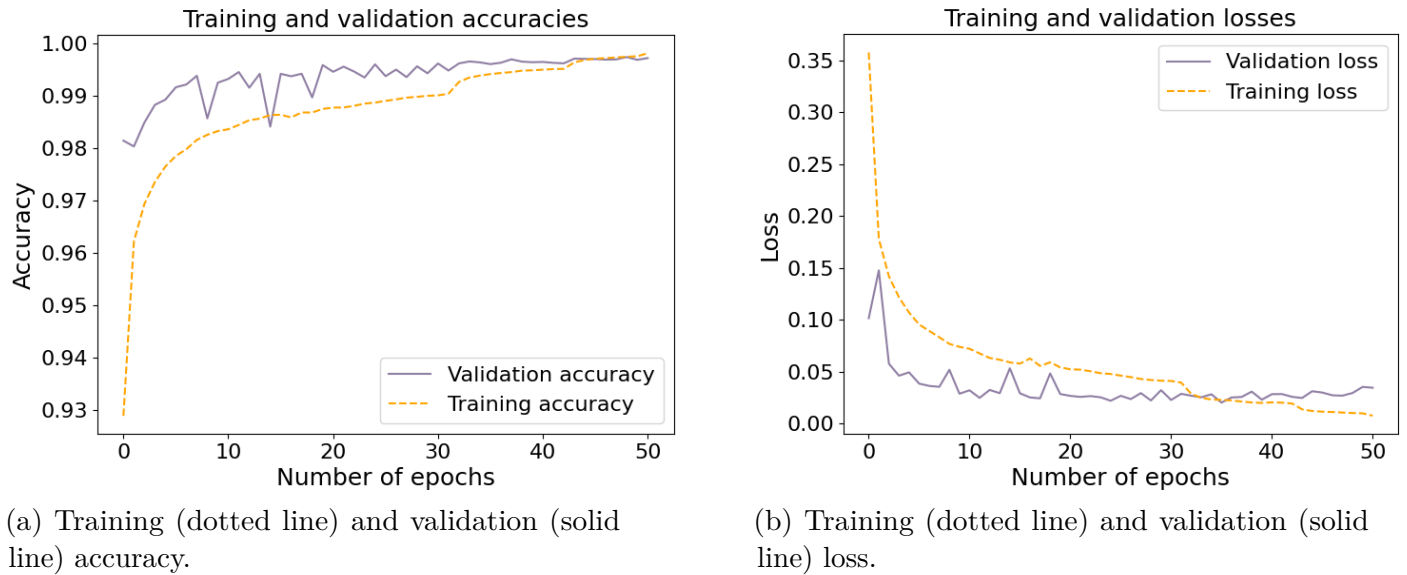


Figure 5.11: Accuracy and loss plots of the optimized baseline model.

The optimized baseline model reaches 99.64% of accuracy on the **test set**, leading to the following confusion matrix.

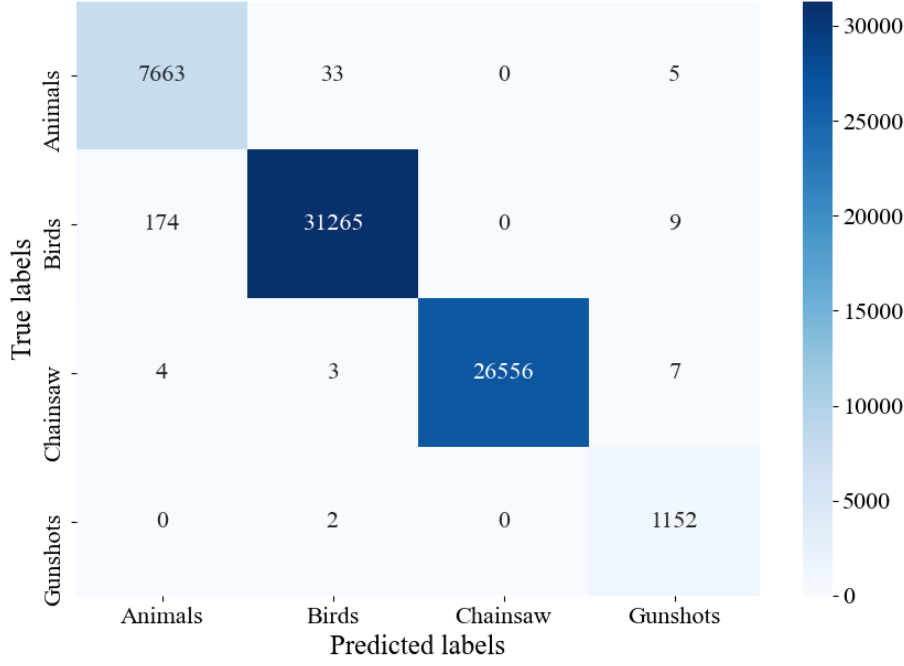


Figure 5.12: Confusion matrix of the optimized baseline model on the test set of the *Events* dataset.

Finally, the precision, recall, and F1-score for each class with this optimized baseline model are shown in Table 5.5.

	Animals	Birds	Chainsaws	Gunshots	Weighted average
Precision	0.977	0.999	1	0.982	0.997
Recall	0.995	0.994	0.999	0.998	0.996
F1-score	0.986	0.996	1	0.99	0.996

Table 5.5: Metrics analysis of the baseline optimized model in terms of precision, recall, and F1-score on the test set.

5.4 Summary

In this chapter, we have described the training setup with sufficient detail to make the results reproducible. To that purpose, an explanation of the used machine, the libraries, and the general hyperparameters have been given. Then, we analyzed the **MobileNetV1** architecture, chosen as our baseline model, because the targeted classification task has never been done in the literature, making it impossible to compare our results with state-of-the-art models. Concretely, we chose this architecture because it is a powerful one that has been studied, especially for embedded devices. It uses depthwise separable convolutions instead of normal convolutions, allowing a computational complexity gain of $\frac{1}{N} + \frac{1}{D_K^2}$ where N is the number of output channels and D_K is the height-width of the filter. Finally, several experiments have been conducted to obtain the highest accuracy of the model while reducing as much as possible its number of MACC operations to reduce the energy consumed for an inference :

1. The impact of the number of training samples on the validation accuracy has been studied. It allowed to choose only **5%** of the *Events* dataset for the remaining experiments due to the high

accuracy ;

2. A study of the best feature type has been performed. Among 178 variations of the seven features presented in Section 2.6.3, the **MFCC feature with a Hanning window of 512 samples, 75% overlap, 128 mel filters, and 39 mel filter cepstral coefficients** was chosen. The choice comes from three main aspects : its complexity, its number of pixels in the input slice, and its validation accuracy. Firstly, the time needed to compute the feature was among the smallest, with approximately **2 ms** on the setup machine. Secondly, the number of pixels of this feature is equal to **1599** (39×41), which is one of the smallest among all feature variations. It allowed a reduction of the number of MACC operations of *MobileNetV1* initially from **572,821,068** with an input image of shape **(224, 224, 3)** to **13,026,380** with an input shape of **(39, 41, 1)**. Finally, the validation accuracy obtained by the baseline model with this feature set lay on the Pareto front in terms of validation accuracy and number of MACC operations. It yielded a validation accuracy higher than **0.96** ;
3. The impact of the data augmentation pipeline was analyzed. Performing **Time Shift**, **Noise Injection**, **Pitch Shift** and **Time Stretch** on the training dataset allowed to boost the validation accuracy significantly by **1.19%** on average. More specifically, the animals accuracy has been increased by **5.5%**, the birds class by **1.1%**, while the chainsaws class has been negatively impacted by **0.8%** because we did not perform data augmentation on it. The gunshots class has not been improved nor negatively impacted.
4. Hyperparameter tuning was done to increase further the validation accuracy. The search space took into account the batch size, the learning rate, and the optimizer. By using the **Tree-Structured Parzen Estimator** algorithm with 30 trials, we were able to boost the validation accuracy by **1.97%** compared to the previous result (obtained after data augmentation). The tuned hyperparameters are the **Adam** optimizer, a learning rate with a value equal to **0.001** and a batch size set to **128** ;
5. The results obtained in all experiments were transferred for the last experiment, which carries the training on the entire *Events* dataset. The baseline model is able to reach **99.64%** of accuracy on the test set for a number of MACC operations equal to **13,026,380**. The performance we get is thus close to the ideal case.

CHAPTER 6

MODEL REFINEMENT FOR MICROCONTROLLER DEPLOYMENT

The performance obtained by the baseline model on the test set is close to the ideal case. However, the model is too big and too energy-demanding to fit on the small ultra-low-power Apollo3 Blue embedded device that is chosen for this work. The *MobileNetV1* architecture contains 3 232 388 parameters which require 12 929 552 bytes of ROM and the model inference takes 13 026 380 of MACC operations with the optimized feature set having a shape of 39×41 pixels. However, as explained in this chapter, the chosen embedded device has a ROM limited to 1 MB, and the number of MACC operations for the maximum usage (100%) of the CPU cannot exceed 5 330 000 for a target inference time of maximum 1 second. Therefore, there is a need to obtain a shallower architecture, respecting the requirements of the microcontroller.

To that purpose, Section 6.1 introduces the choice we made for the microcontroller, and Section 6.2 explains in detail why the baseline model cannot fit on the chosen device. The remaining of this chapter introduces different experiments, allowing to obtain a smaller architecture still able to offer a high accuracy on the test set.

6.1 Microcontroller selection

The choice of a microcontroller for a specific application is important as we are always more interested in some characteristics than others. In this work, we want to implement an ultra-low-power device. Our choice should therefore analyze the active current of the microcontroller. However, we also need to have enough memory to make our application work. Finally, we need to choose an ARM Cortex-M based processor as we want to use the *CMSIS-NN* library [14], which is designed for this specific kind of core. Note that the *CMSIS-NN* library is fully explained in Chapter 7.

Here follows a non-exhaustive list of known and interesting microcontrollers for our application with some of their important characteristics :

Device	Provider	Core	Maximal Frequency [MHz]	Active Current [$\frac{\mu A}{MHz}$]	RAM [kB]	ROM [kB]
STM32L072x8	STMicroelectronics	ARM Cortex-M0+	32	93	20	192
STM32L476xx	STMicroelectronics	ARM Cortex-M4	80	39/100	128	1024
LPC55S6x	NXP Semiconductors	ARM Cortex-M33	150	35/75	320	640
Apollo2	Ambiq Micro	ARM Cortex-M4	48	<10	256	1024
Apollo3 Blue	Ambiq Micro	ARM Cortex-M4	96	<6	384	1024
<i>Apollo4 Blue</i>	<i>Ambiq Micro</i>	<i>ARM Cortex-M4</i>	<i>192</i>	<i><3</i>	<i>1843</i>	<i>2048</i>

Table 6.1: Possible devices for our application with their specific global characteristics. All devices are based on a 32-bit RISC architecture. All cores are ARM Cortex-M based because we want to use *CMSIS-NN*. For the active current, when two values are given, it is the theoretical lowest one on the left and the biggest one on the right, depending on the chosen CPU frequency. The device in orange is a really good candidate but was not available at the time of this thesis.

The two devices on the top of the list are candidates from previous works focusing on environmental sound classification [176, 30]. The three candidates that follow are known by the scientific community and have good characteristics. *Ambiq Micro* is a company that focuses on building ultra-low-power devices for IoT applications. By looking in detail in Table 6.1, we see the astonishing numbers for the active current compared to the other devices. Furthermore, the two types of memory are large for the application we target. It is thus not a surprise that we decide to choose the *Apollo3 Blue* microcontroller. Note that we do not require a too big core frequency in this application since the final network will be designed to be computationally light. Also note that the board from the *NXP Semiconductors* provider uses an ARM Cortex-M33, which does not benefit from the optimizations of *CMSIS-NN* that are specific to the Cortex-M4 and Cortex-M7 cores [177]. In the following Table is summarised more characteristics of the chosen device :

Device	FFT HW Accelerator	Floating Point Unit (FPU)	Cache	ADC
Apollo3 Blue	False	True	True (16 kB)	True (14 bits)

Table 6.2: More detailed characteristic of the chosen device.

As it can be seen in Table 6.2, the microcontroller contains a floating-point unit which allows being efficient when working with floating-point numbers. However, it does not contain an FFT hardware accelerator which would be really useful for the onboard feature processing part since the transformation of an audio to an image has been shown in Section 2.6.3.1 to use many FFT operations. Fortunately, the device contains a small cache. It is helpful in terms of latency and energy efficiency since it allows not to fetch data and instructions from the onboard memory. Finally, an ADC of 14 bits is available on the board. However, except for the cache, these small features are secondary as they are not used in this work.

6.2 Impossibility of the baseline model on the Apollo3 Blue microcontroller

The chosen microcontroller has a Cortex-M4F core running at a frequency of 48 MHz. The company STMicroelectronics estimates that an ARM Cortex M4F based device uses approximately 9 cycles per MACC operation (including the load and write operations) [178]. At a frequency of 48 MHz, it is approximately 5.33 million MACC per second at 100% CPU utilization. Because we work on 1-second slices, this metric is of interest. However, we want an ultra-low-power model, and the inference is not the only process happening on the board. The feature processing is also an inherent part of it. A threshold of a maximum of 10% of CPU utilization is thus fixed for the model inference. The inference taking place each second must therefore compute at most 533 000 MACC operations. Because the optimized baseline model takes 13 026 380 of MACC operations for a single inference, we demonstrate that it is impossible to respect our requirements. The optimized baseline model requires more resources than those we have at our disposal to infer each second, leading to a non-real-time application. Moreover, the optimized baseline model cannot fit on the microcontroller, as it contains 3 232 388 parameters, leading to a ROM usage of 3.232 MB if the weights are quantized with 8 bits. However, the ROM of the microcontroller does not exceed 1 MB, making the baseline model impossible to fit on the Apollo3 Blue microcontroller. We decide to look for an architecture requiring less than 10% of ROM usage to minimize flash access and let space for improvement.

6.3 Experiments flow

This section presents a flow of specific experiments done to obtain a smaller architecture able to preserve a high accuracy on the *Events* dataset. The diagram flow of the experiments is shown in the following figure. Unless it is mentioned, the training setup is the same as the one presented in Section 5.1, while the used dataset is the same as the one presented in Section 5.3.

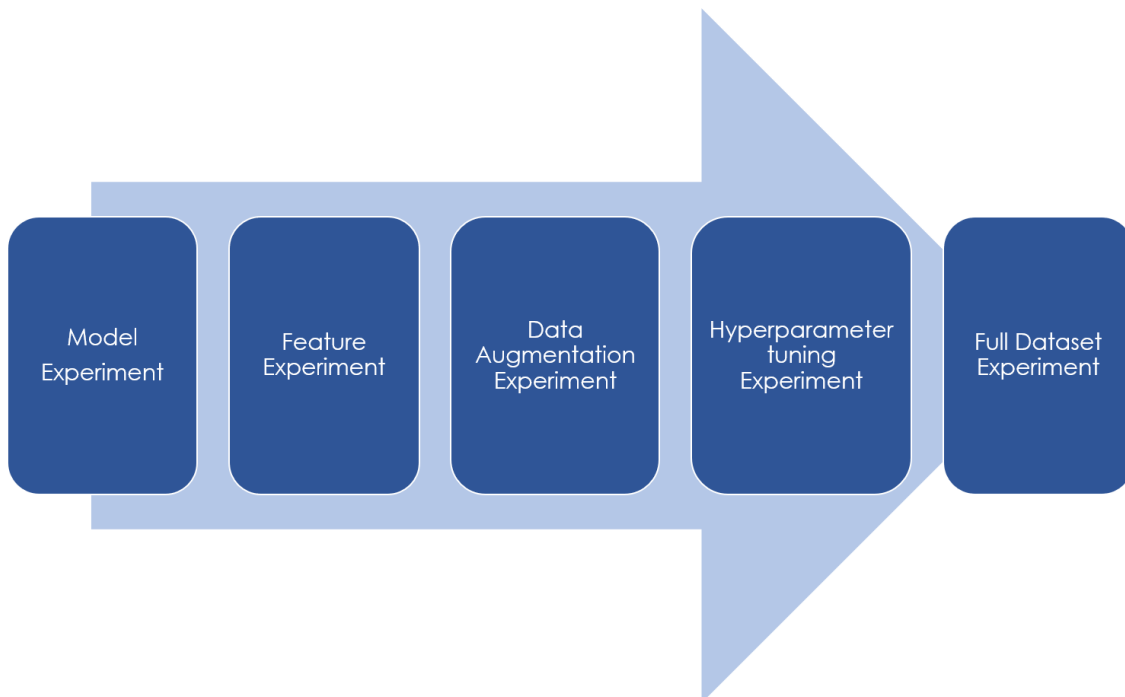


Figure 6.1: Diagram flow for all the experiments done to refine the baseline model.

6.3.1 Obtention of the model architecture

The purpose of this experiment is to find a shallower architecture, requiring less computation to perform the classification task with high accuracy. There are many architectures of convolutional neural networks, some of them being more energy-demanding because of the type of layers or the network depth. In this work, we have decided to compare several known architectures and derive novel ones based on the originals. Of course, the studied models are not the most energy-demanding since the purpose is to limit the number of computations of the model as much as possible. Furthermore, we have modified the softmax layer of every architecture with 4 output neurons to fit our classification task.

- All the typical smaller versions of *MobileNetV1*, known as *0.75-MobileNetV1*, *0.5-MobileNetV1* and *0.25-MobileNetV1*. These reduced versions depend on the α factor in the architecture, referred to as the width multiplier and discussed in Section 5.2.
- *LeNet5* architecture, by its creator Yann LeCun, is the first convolutional neural network used for handwritten and machine-printed character recognition. Its architecture is straightforward : it consists of two sets of convolutional and average pooling layers, followed by a flattening convolutional layer, then two fully connected layers and finally a softmax classifier. It is represented in Figure 6.2 where each layer parameter can be derived from the original image.

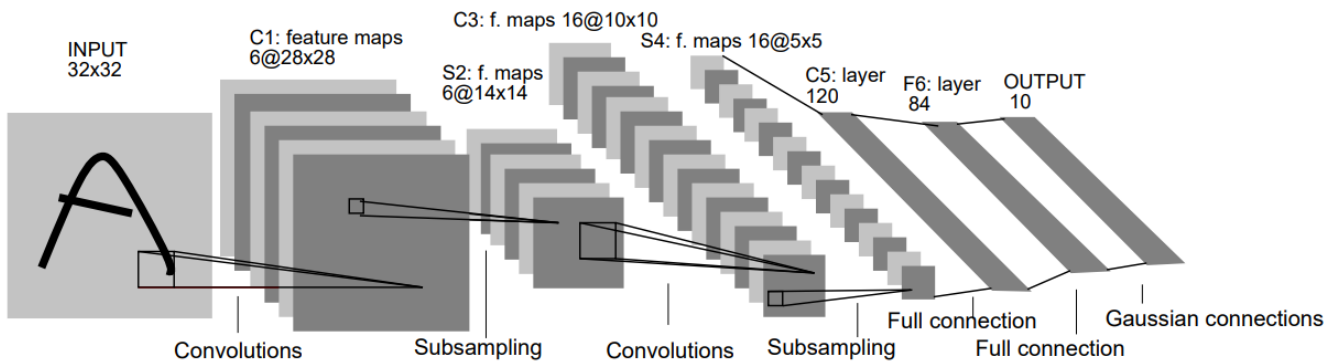


Figure 6.2: LeNet5 architecture [24]

We have also implemented a modified version of this architecture, which we call *LeNet5-M* that modifies the number of filters in each layer. Only 3 feature maps are used for the first convolutive layer and 8 for the second while we change the number of units in the first fully connected layer to 12 instead of 84.

- The *Strided-DS 24* architecture created by Jon Nordby [30]. Concretely, the name comes from the fact that the architecture uses strided depthwise separable convolutions. The stride of size 2 in the depthwise separable convolutions allows replacing max-pooling layers. This architecture has been designed to reduce the number of operations of the model and is thus interesting for our experiment. The architecture is shown in Figure 6.3.

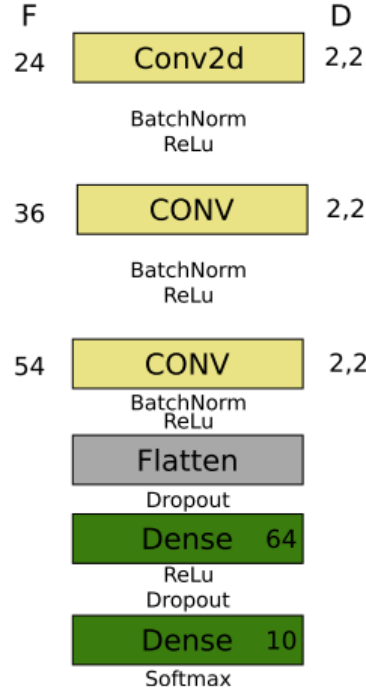
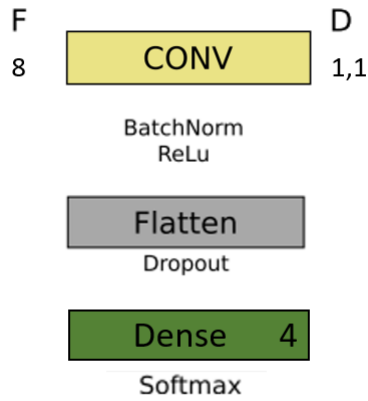
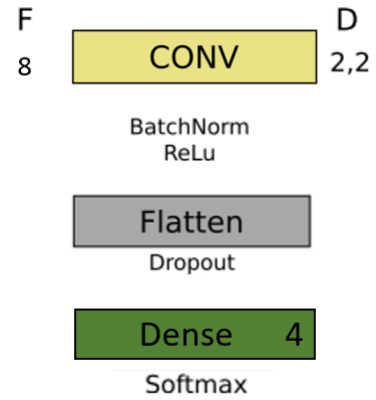


Figure 6.3: *Strided-DS 24* architecture [30]. The keyword 'F' designates the number of filters or feature maps in each layer, whereas the keyword 'D' is the stride size in each layer. **Conv2d** refers to a normal convolution layer, whereas **CONV** is a depthwise separable convolution layer. Padding is used in the architecture. A kernel size of (3,3) is used for each layer, and the dropout factor is 0.5.

- 6 variations of the *Strided-DS 24* architecture have been analyzed in this work. These are called *DSC-L1*, *DSSC-L1*, *DSSC-L2*, *DSSC-L3*, *DSSC-L4*, and *DSSC-L5*. The architecture of *DSC-L1* is shown on the left of Figure 6.4, while the architecture of *DSSC-L1* is shown on the right of the same figure. Concretely, *DSC-Lx* refers to a network with x layers of depthwise separable convolutions, and *DSSC-Lx* refers to a network with x layers of strided depthwise separable convolutions.



(a) DSC-L1 architecture.



(b) DSSC-L1 architecture.

Figure 6.4: The keyword 'F' designates the number of filters or feature maps in each layer, whereas the keyword 'D' is the stride size in each layer. **CONV** refers to a depthwise separable convolution layer. Padding is used in the architecture. A kernel size of (3,3) is used for each layer and the dropout factor is 0.5.

The architectures of *DSSC-L2*, *DSSC-L3*, *DSSC-L4*, and *DSSC-L5* are straightforward when knowing the one of *DSSC-L1*. Each new convolutive layer is the same as the previous one except that four feature maps are added.

By training all these different networks, we can compare in Figure 6.5 the validation accuracy, the number of MACC operations, and the number of parameters obtained with each architecture.

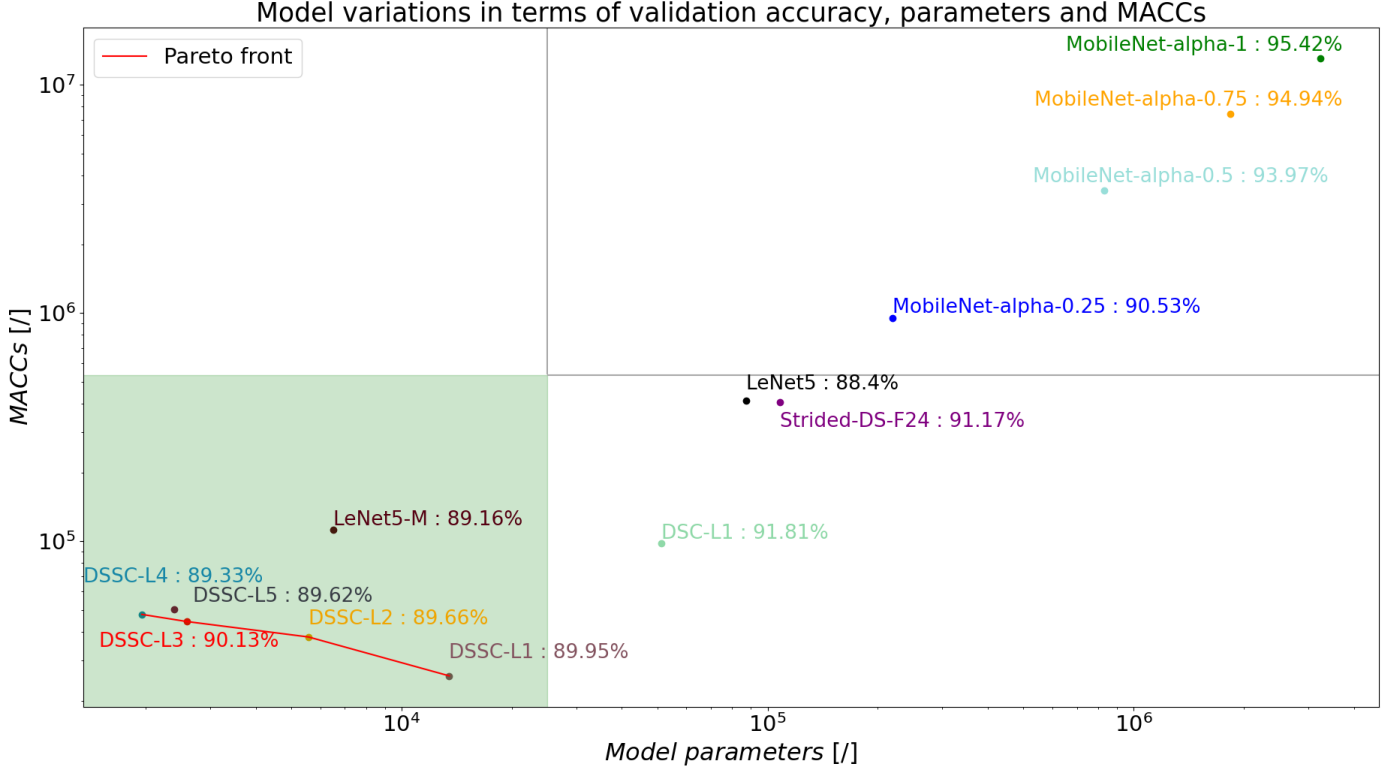


Figure 6.5: Each studied network is shown in a plot : model parameters versus the number of MACC operations with their specific validation accuracy. The green area corresponds to the requirements we fixed in Section 6.2. The Pareto front is represented in a solid red line.

The model that is chosen is the *DSSC-L4* architecture since it is one of the best compromises we can find in terms of MACC and the number of parameters. Moreover, the loss of accuracy with this model is acceptable : approximately 6% compared to the *MobileNetV1* optimized baseline model. Choosing this *DSSC-L4* architecture compared to *MobileNetV1* allows reducing the number of MACC operations by a factor of 273 and the number of parameters by a factor of 1655.

6.3.2 Extraction of the best feature type

The best feature set has been found on the baseline model in Section 5.3.2. However, due to the architecture, the *MobileNetV1* model requires that the feature size has at least 32 pixels in height and in width. This is due to the downsampling factor of the architecture equal to 32, as highlighted in Figure 5.1. The feature extraction part of the baseline model containing the convolution layers has an input image of shape (224, 224, 3), while the 1024 feature maps before the average pooling layer are of size (7, 7). Therefore, the square image, composed initially of 224 pixels in the height and width dimensions, is processed up to a square feature map of 7 × 7 pixels at the end of the backbone. This factor of size reduction equal to 32 for *MobileNetV1* is named the downsampling factor. In the

DSSC-L4 architecture found for the refined model, the downsampling factor is equal to 13. Therefore, it lets place to improvement in terms of the number of pixels in the input slice, which plays a role in the energy-efficiency of the model as explained in Section 5.3.2.

Ten more variations of the chosen *MFCC* feature, which are represented in Table 6.3, are experimented. The result is shown in Figure 6.6. The initial feature set, found for the optimized baseline model, is called *MFCC_0*. We see that it is possible to have a lower number of MACC operations and a shorter processing time (on the setup machine) for a small loss of validation accuracy with the *MFCC_4* feature set. This new variation is thus chosen for the refined model.

Name	FFT size	Overlapping size	Number of mel filters	Number of coefficients
MFCC_0	512	384	128	39
MFCC_1	512	512	128	39
MFCC_2	1024	384	128	39
MFCC_3	1024	512	128	39
MFCC_4	1024	1024	128	39
MFCC_5	512	384	128	13
MFCC_6	512	512	128	13
MFCC_7	1024	384	128	13
MFCC_8	1024	512	128	13
MFCC_9	1024	1024	128	13

Table 6.3: Feature variations studied for the best feature set on the refined model.

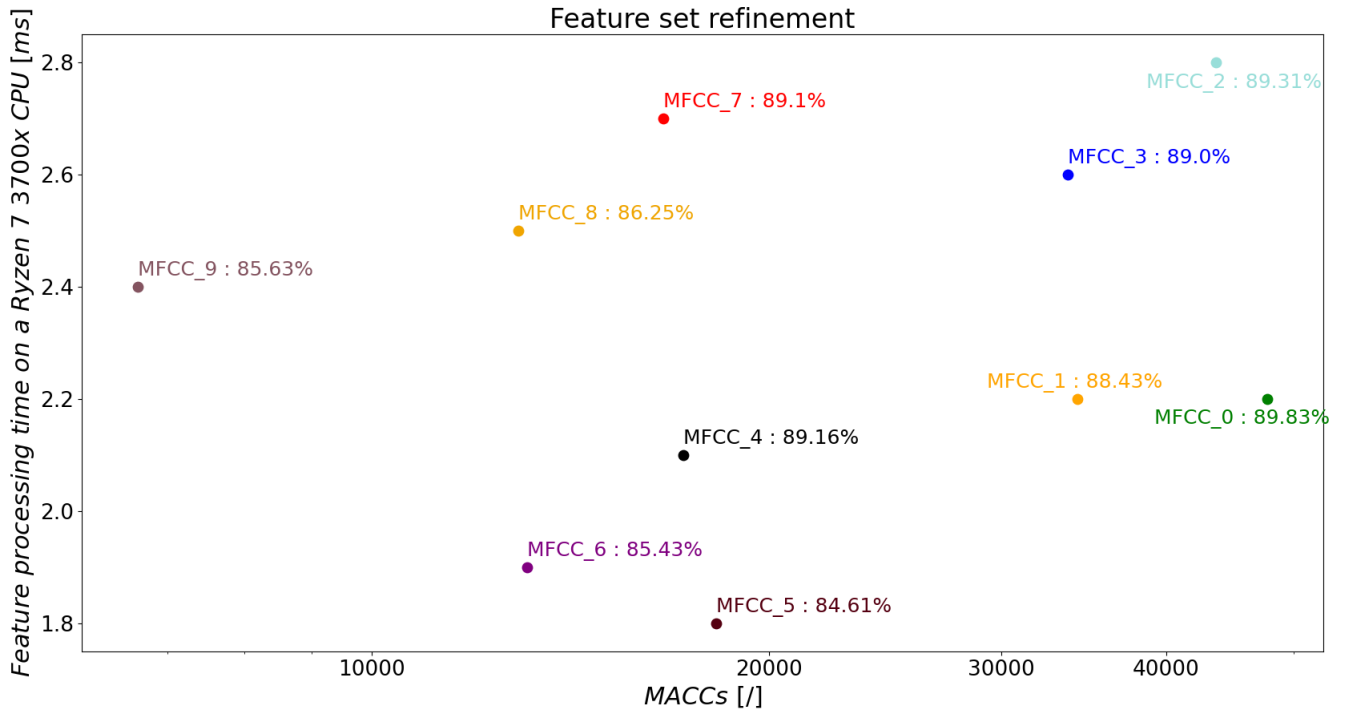


Figure 6.6: 10 variations of the MFCC feature found for the baseline model (*MFCC_0*) are studied in terms of the number of MACC operations, the complexity, and the validation accuracy.

6.3.3 Data augmentation technique

Taking the same data augmentation ensemble as the one used for the baseline model is not a good idea without experimenting. Even if the dataset that is used is the same, the network is less complex, and the feature type is different. The same conclusions cannot thus be made without making the trial. The experiment that follows is similar to the one done in Section 5.3.3. However, we do not perform the **Frequency & Time Masks** spectral data augmentation technique. Indeed, even if the accuracy is higher with it than with another technique, it requires too many training samples, and the training process would be too long to handle afterward. We show the result of the data augmentation experiment in Figure 6.7.

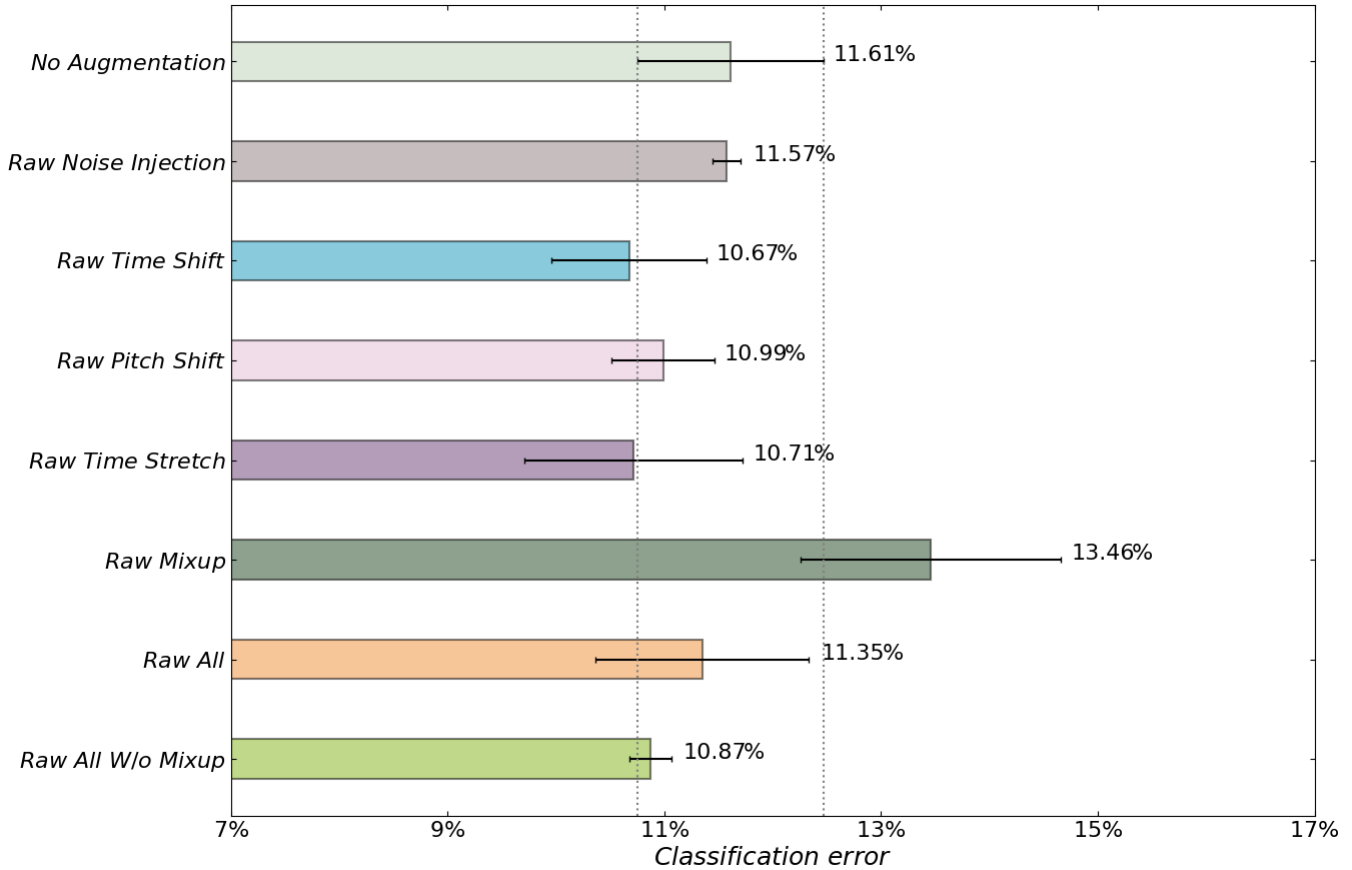


Figure 6.7: Classification error on the validation set for different augmentation methods. Bars and whiskers indicate the mean and its 95% confidence interval computed from five repetitions of each experiment.

Similar conclusions can be made with respect to the baseline model. Two main differences are still noticeable :

- The *Time Stretch* feature now has a much bigger confidence interval ;
- The combination of all raw features does not significantly reduce the classification error.

Nonetheless, we still choose the *Raw All W/o Mixup*. Even though the mean validation accuracy is not the best, its confidence interval is tiny, proving the efficiency of this set. The difference in the confusion matrix before and after the data augmentation experiment is shown in Figure 6.8. To that purpose, we took the worst-performing models without and with data augmentation.

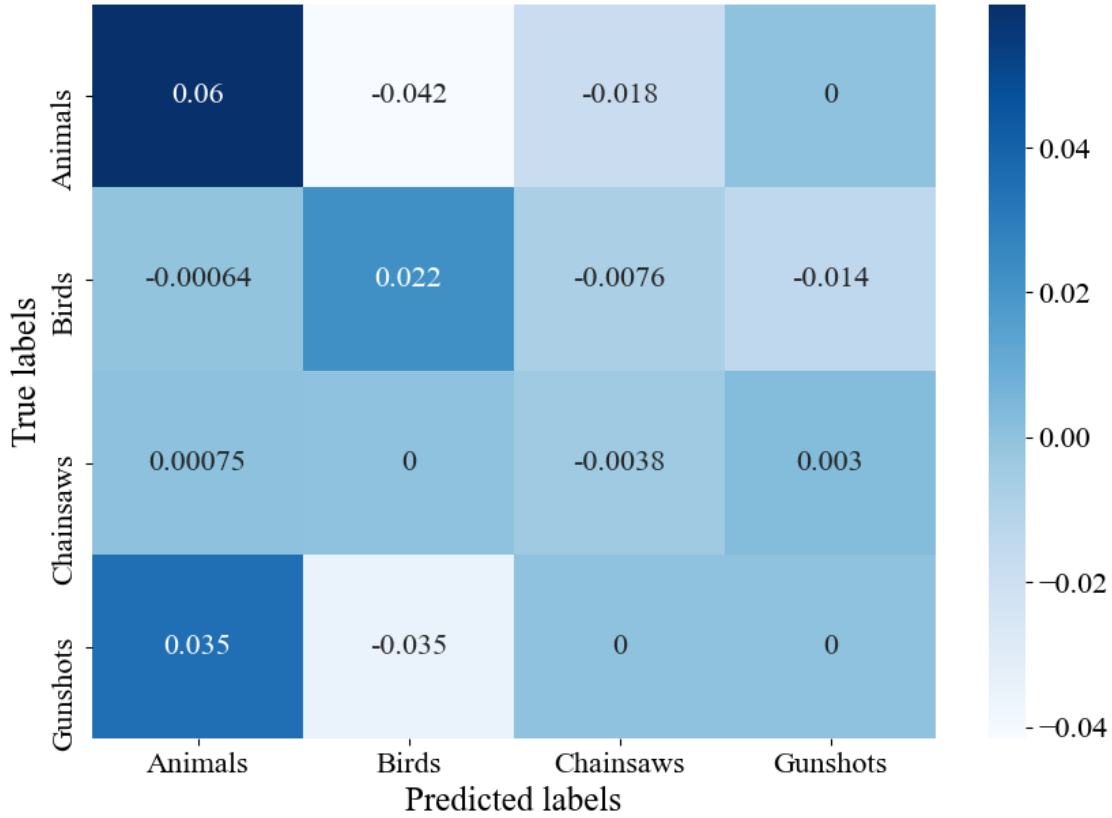


Figure 6.8: Difference of confusion matrices with and without data augmentation. Positive values indicate a positive change in confusion, and negative values indicate a negative change in confusion.

As for the baseline model, the validation accuracy does not improve significantly, but the animals classification is boosted by 6% and the birds class by 2.2%. The chainsaws class is negatively impacted by 0.38%, while the accuracy is unchanged for the gunshots class. However, for the latter, the false positives have increased for the animals class compared to the birds class.

6.3.4 Hyperparameter tuning

This experiment relies on the same basis as the one done in Section 5.3.4. However, we also study particular parameters of the *DSCC-L4* architecture. We play on the number of feature maps for each convolution layer. We set values that allow staying below 22 000 MACC operations of the refined model. The entire search space is shown in Figure 6.9 and contains 2880 states. We perform the hyperparameter search with the *Tree-Structured Parzen Estimator* algorithm using 70 trials. The final validation accuracy result is given in Table 6.4 as well as the result obtained just after the data augmentation experiment. We obtain a significant improvement of 5.14% on the classification accuracy when using the worst performing model after data augmentation.

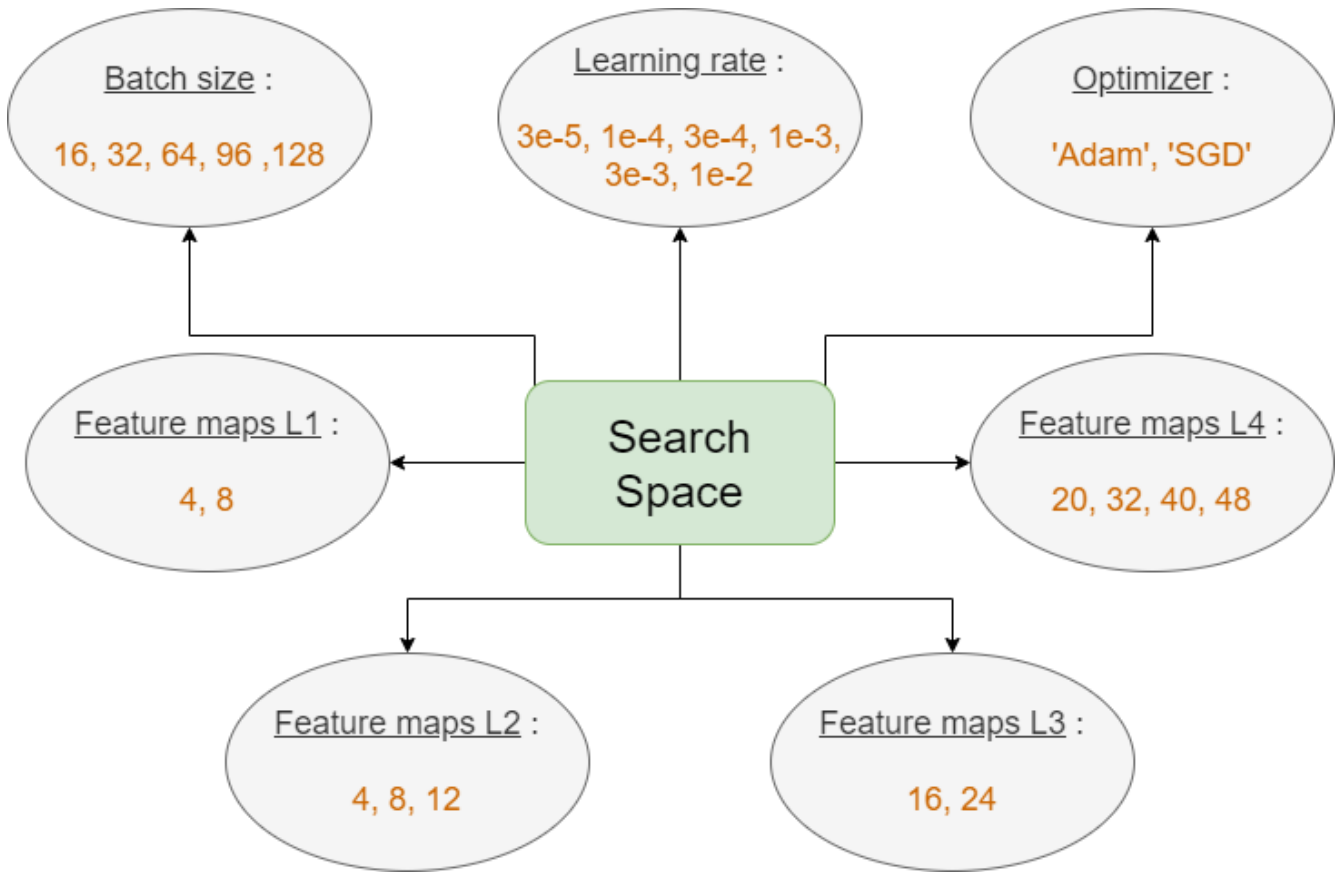


Figure 6.9: Search space for the hyperparameter tuning experiment. **Feature maps Lx** represents the number of feature maps in the depthwise separable convolution layer number x , taking one of these values $\{1, 2, 3, 4\}$.

Before hyperparameter tuning	88.81%
After hyperparameter tuning	93.95%

Table 6.4: Validation accuracy of the refined model before and after hyperparameter tuning.

The final hyperparameters are: an Adam optimizer, a batch of 96 samples, and a learning rate equal to 0.01. The parameters of the architecture are 8 feature maps for the two first convolutive layers, 16 for the third one, and 40 for the last one. This specific architecture allows reducing the number of MACC operations of the model up to 16 344 while significantly improving the classification accuracy.

A slice plot is represented in Figure 6.10. It demonstrates the considerable impact hyperparameter tuning has on neural network performance. The worst performing model reaches a validation accuracy of 85.2%, while the model with the best hyperparameters achieves almost 94%. The slice plot allows presenting the search space exploration.

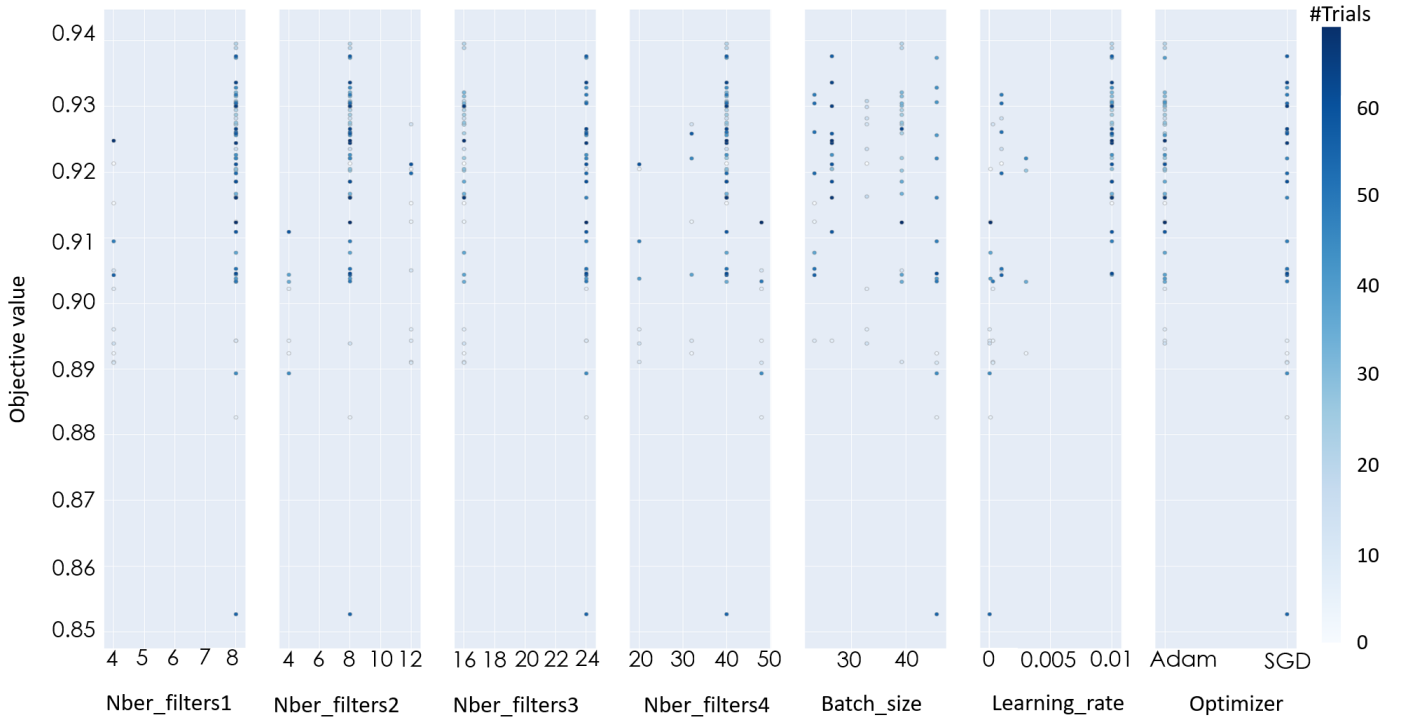
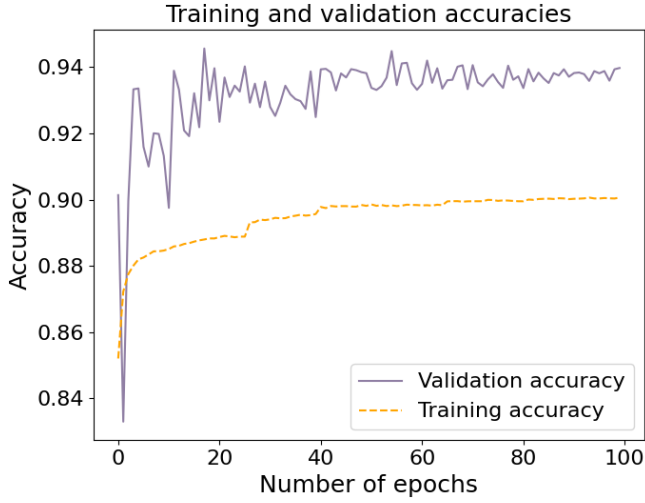


Figure 6.10: Hyperparameter tuning search space exploration of the refined model. **Nber_filtersX** represents the number of features map in layer **X**.

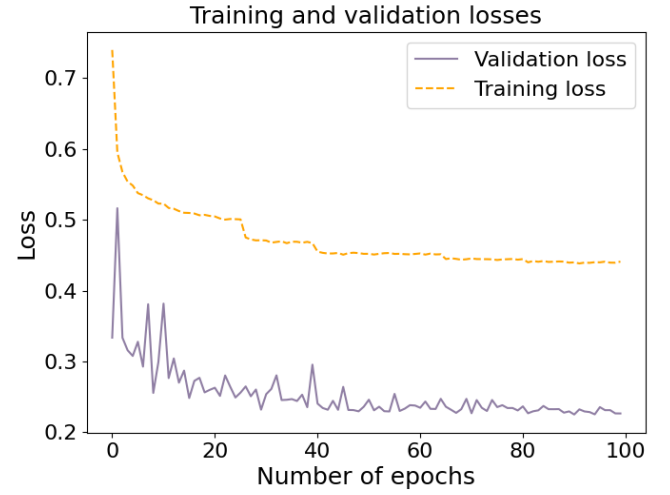
6.3.5 Optimized refined model

We can now use the results of the previous experiments for the training of the refined model on the complete dataset. Therefore, we train the *DSSC-L4* modified architecture, due to the hyperparameter tuning experiment, with 2 359 816 MFCC images, due to the data augmentation experiment. The MFCC images are computed with the following hyperparameters : a Hanning window of 1024 samples, 0% overlap, 128 mel filters, and 39 mel filter cepstral coefficients. The training hyperparameters are a high learning equal to 0.01, a batch size of 96 samples, and the Adam optimizer. The same callbacks as in Section 5.3.4 are used for the training.

The training is stopped manually after 100 epochs and gives the results shown in Figure 6.11. As seen, the difference between validation and training accuracies is significant (4%). This is due to the dropout layer. During training, it sets 50% of the fully connected units to 0. The information from 60 of the 120 units of the fully connected layer is not used. Therefore, it is logical that the classification capacity is largely impacted. In the final training of *MobileNetV1* discussed in Section 5.3.5, the dropout rate is significantly lower (0.1%), explaining why the accuracy drop is much smaller. The accuracy plot demonstrates an instability in the first half of the training due to the large learning rate. It stabilizes along with the training with a reduction of this learning rate up to 0.000625. If we had continued this training up to a large number of epochs, which was not done due to the time it takes, the stabilization process would have probably continued with a further decrease of the learning rate. Interestingly, the optimized refined model is not positively impacted by an increase in the training dataset. Indeed, the result after hyperparameter tuning is an accuracy of 93.95% on the 5% *Events* validation dataset, while it is only 93.5% when using the complete validation dataset. Even though the training and validation datasets have increased in the number of samples, the generalization has not improved.



(a) Training (dotted line) and validation (solid line) accuracy.



(b) Training (dotted line) and validation (solid line) loss.

Figure 6.11: Accuracy and loss plots of the optimized refined model.

The confusion matrix of the optimized refined model on the complete test set is shown below. Moreover, we present the precision, recall, and F1-score metrics in Table 6.5.

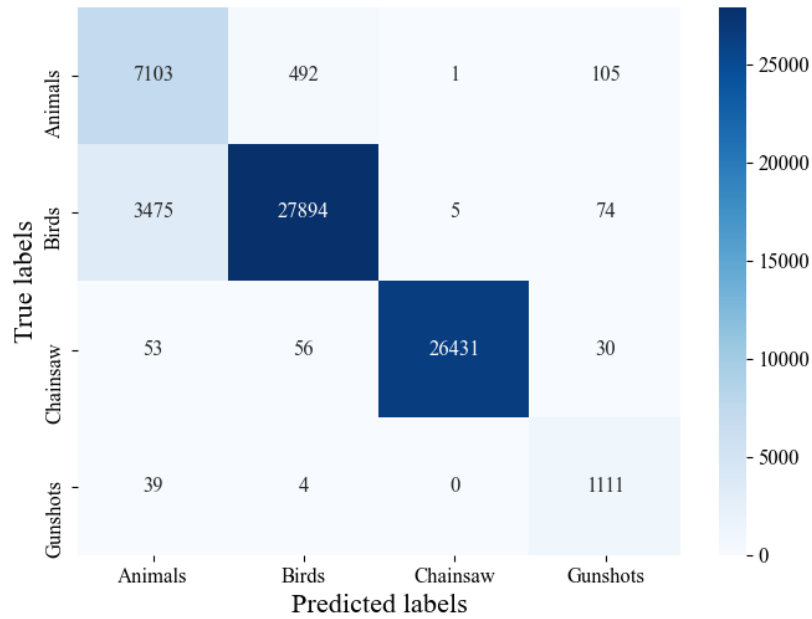


Figure 6.12: Confusion matrix of the optimized baseline model on the test set of the *Events* dataset.

	Animals	Birds	Chainsaws	Gunshots	Weighted average
Precision	0.666	0.981	1	0.842	0.95
Recall	0.922	0.887	0.995	0.963	0.935
F1-score	0.773	0.931	0.997	0.898	0.939

Table 6.5: Metrics analysis of the refined optimized model in terms of precision, recall, and F1-score on the test set.

6.4 Summary

In this chapter, we have studied various known microcontrollers for our audio classification task. The Apollo3 Blue embedded device from Ambiq Micro has been chosen due to its ultra-low-power performance and its large memory capacities. Moreover, the microcontroller is based on an Arm Cortex-M processor, which is essential to use the *CMSIS-NN* library for the embedded implementation in the next chapter.

We have demonstrated that the optimized baseline model from Chapter 5 is too complex and too power-consuming to fit on the chosen microcontroller. Indeed, the architecture requires 13 026 380 of MACC operations for one inference and 3.232 MB of Flash for storing the weights if they are quantized with 8 bits. However, the Apollo3 Blue microcontroller is able to support 5 330 000 MACC operations per second at a maximum CPU usage, and it contains only 1 MB of Flash memory.

We therefore performed a set of experiments to obtain a shallower architecture preserving a high accuracy on the *Events* dataset. We targeted a maximum CPU and ROM usage of 10% for the real-time 1-second classification task on the microcontroller :

- Multiple original and novel architectures were studied. A model based on four depthwise separable convolutions was chosen. Compared to the baseline model, it gave a reduction of MACC operations and model parameters respectively by a factor 273 and 1655 for a validation accuracy loss of 6.09% ;
- Due to the finer downsampling factor (13) of the chosen *DSSC-L4* architecture, it was possible to find an MFCC feature yielding less model MACC operations. Compared to the optimized feature set of the baseline model, the optimized MFCC feature of the refined model allowed a reduction of the number of MACC operations by a factor of 2.77 for a validation accuracy loss below 1% ;
- We perform a data augmentation experiment to boost the classification accuracy of the refined model. Choosing **Raw All W/o Mixup** allowed a 6% boost for the animals class, and 2.2% for the birds category ;
- A study of the training hyperparameters, as well as some architectural parameters was made to decrease further the classification error. Among the 2880 studied states, the algorithm converged for an Adam optimizer, a learning rate value of 0.01, and a batch size of 96 images. Furthermore, the number of feature maps of each layer has been optimized in the process, allowing at the end a decrease of the number of MACC operations from 17 222 to 16 344. The tuning allowed a considerable validation accuracy boost of 5.14% ;

The results of these experiments was then used to train the refined architecture on the complete *Events* dataset. The model achieves 93.5% on the test set. It thus suffers from a test accuracy loss of 6.14% compared to the optimized baseline model. However, it allowed a reduction of the number of MACC operations by a factor of 797 and a decrease of the number of parameters by a factor of 1605. These significant improvements allow an embedding of the refined convolutional neural network on the Apollo3 Blue microcontroller.

CHAPTER 7

IMPLEMENTATION OF THE CONVOLUTIONAL NEURAL NETWORK ON AN ULTRA-LOW-POWER MICROCONTROLLER

This chapter introduces the work done to go from a *Keras* model to a *CMSIS-NN* implementation deployable on a microcontroller. First, we mention the software used to create and debug the application. Then, we detail the description and objective of the *CMSIS-NN* library. Afterward, we explain the numerous steps required to convert the network. Finally, we show the network metrics on the chosen microcontroller. The results include the test set accuracy, the runtime execution of the neural network, and the power consumption of the entire device.

7.1 Used Software

The Apollo3 Blue microcontroller is supported by the *ARM Keil MDK* software development tool [179] used in this work. Plenty of other integrated development environments exist, such as *ARM Development Studio* or *Eclipse*. However, only a few support the J-LINK debugger included originally in the Apollo3 Blue microcontroller. For this reason and due to the rich documentation of ARM, the *ARM Keil MDK* software development tool is chosen. We use the *MDK-Professional* version because the free *MDK-Lite* version supports only 32 kB of code size. We use the *uVision5* version for the integrated development environment.

7.2 *CMSIS-NN* library

CMSIS-NN [14] is a set of optimized software kernels developed by ARM [180] for deploying neural networks on Arm Cortex-M processors. In other words, it is an optimized C library created to implement energy-efficient neural networks for intelligent IoT devices.

Concretely, neural network inference based on the *CMSIS-NN* kernel involves a $4.6\times$ improvement in latency (runtime) and a $4.9\times$ improvement in energy efficiency over the baseline *CMSIS-DSP* functions. These numbers, available in [14], have been obtained for a convolutional neural network targeting the CIFAR-10 image classification task [181]. This improvement is mainly due to the use of SIMD instructions for the Arm Cortex-M4 and Cortex-M7 processors. The other benefits come from minor optimizations of the *CMSIS-NN* functions. A SIMD instruction, also known as Single Instruction Multiple Data, allows performing the same operation on several data samples with one instruction. For example, 16-bit Multiply-and-Accumulate SIMD instructions allow accelerating

matrix multiplication used almost in every *CMSIS-NN* function [182]. Indeed, the convolution functions use a specific procedure known in the literature as the *im2col* algorithm. It allows converting the convolution into a matrix multiplication. Its principle is relatively simple: it transforms the input image into a matrix, makes a matrix multiplication between the latter and a reshaped version of the kernel, and then uses the *col2im* algorithm to reshape the obtained matrix into an image. The *im2col* algorithm has the advantage of offering a critical speedup factor of more than 200 according to [183] compared to the original convolution process. However, the *im2col* algorithm requires more memory because of the data redundancy. The comparison between a regular convolution operation and the convolution with the *im2col* algorithm is represented in Figure 7.1.

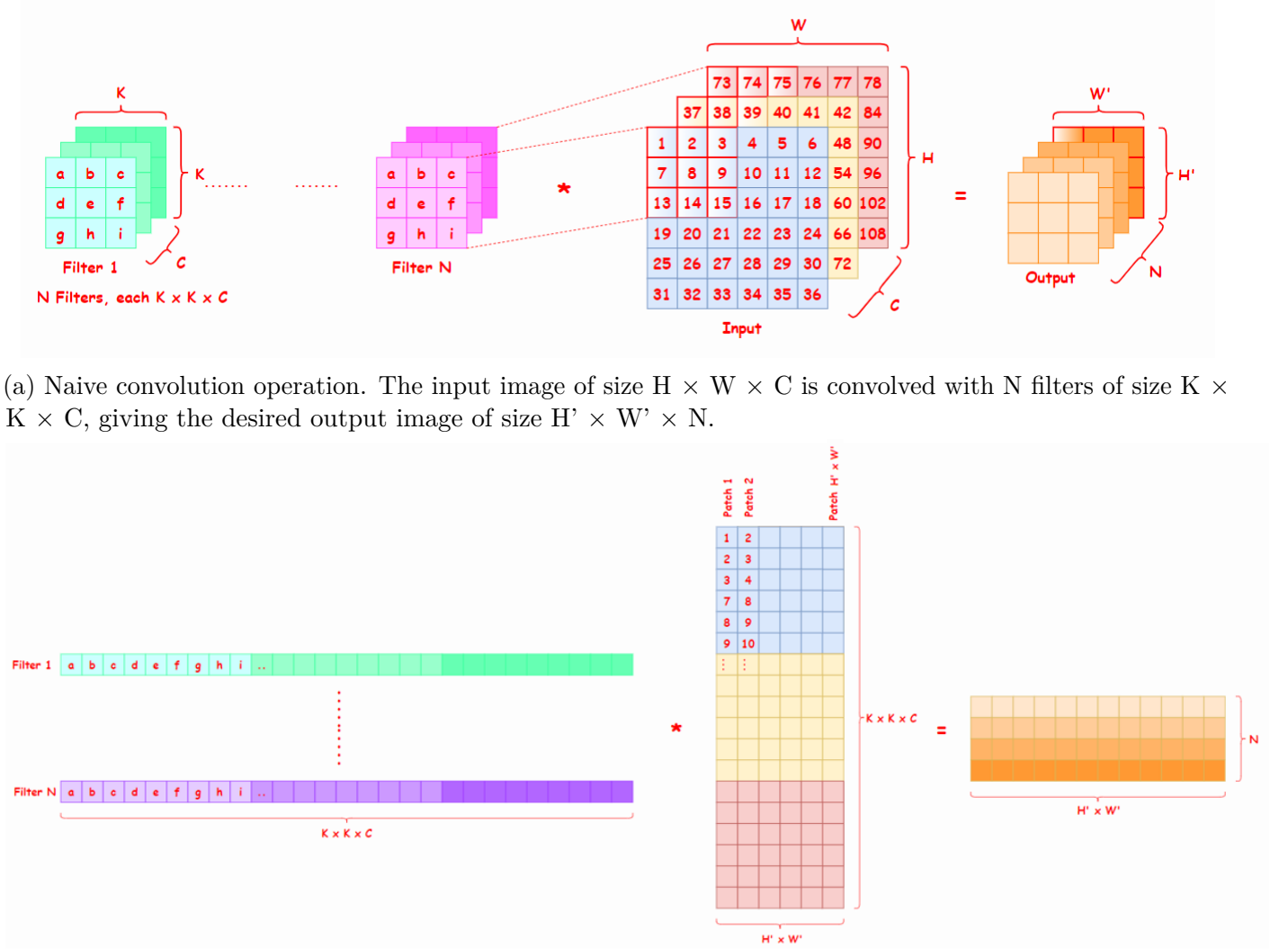


Figure 7.1: Comparison between the naive convolution operation and the optimized convolution using the *im2col* algorithm [184]. Note in this representation that the number of squares for each matrix is not always representative of the reality, whereas the noted dimensions are.

As mentioned in [14], the *CMSIS-NN* kernel is divided into the **NNfunctions** and the **NNSupportFunctions**. The first part contains all the important functions used to implement popular neural network layer types, such as convolution, depthwise separable convolution, fully connected, pooling and activation. The other part contains the utility functions, which are used in the **NNfunctions** part. Examples of **NNSupportFunctions** can be data conversion functions. It is important to note that the **NNFunctions** branch can contain different versions of the same function. For instance, ARM developed the functions `arm_fully_connected_q7` and `arm_fully_connected_q7_opt`. Both allow the implementation of a fully connected layer with 8-bit data and weights in a neural network. They achieve the same goal but do not use the same ordering of the weights and do not have the same performance metrics. Indeed, the first one is the basic version, while the latter is an optimized version using a specific reordering of the fully connected weights to obtain a better runtime of the layer. Therefore, if the performance of the neural network is important for the application, it is useful to analyze in detail all the different versions of a function doing the same job. Note also that some *CMSIS-NN* functions have specific requirements on their input and output. If they are not satisfied, the *CMSIS-NN* implementation is replaced by the basic C version, which is not optimized for neural network inference. For instance, the `arm_convolve_HWC_q7_basic_nonsquare` function requires that the number of input channels is a multiple of 4 and the number of output channels is a multiple of 2. This requirement is due to the use of the SIMD instructions. The structure of the *CMSIS-NN* kernel is given in the following figure, while we can find all *CMSIS-NN* functions here [185].

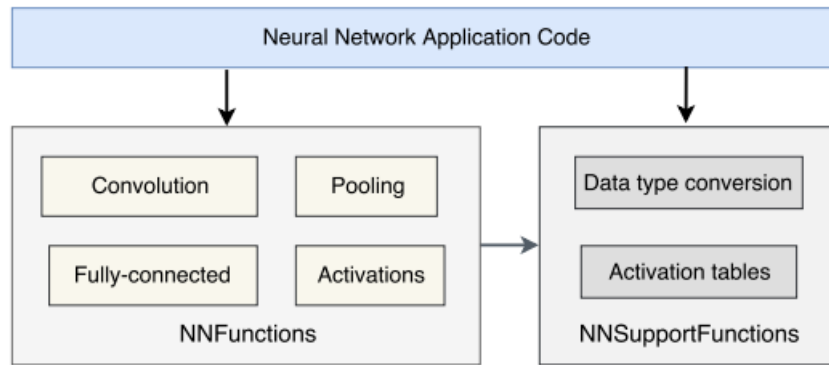


Figure 7.2: Overview of the *CMSIS-NN* kernel structure [14]. The **NNFunctions** part implements the different layers used in neural networks and these functions use utility functions from **NNSupportFunctions**.

The *CMSIS-NN* kernel has been developed using only a full quantization of 8-bit or 16-bit data types. The keyword **q7_t** in the functions name refers to the `int_8t` data type. Several researches showed that neural networks perform well even with low-precision fixed-point representation [186, 187]. Moreover, a neural network initially uses 32-bit floating-point data and weights, but it is not suited for embedded devices since it requires a lot of memory usage and MAC operations are more complicated to compute. In other words, *CMSIS-NN* chose these bitwidths because "fixed-point quantization helps to avoid the costly floating-point computation and reduces the memory footprint for storing both weights and activations, which is critical for resource-constrained platforms" [14].

To deeply understand the process of fixed-point quantization used by the *CMSIS-NN* kernel, we need to introduce the notion of Q format of a number [188]. Concretely, a Q format number is a fixed-point number allowing standard ALU operators to perform fixed-point arithmetic. By convention, it is written **Qm.n** where the letter *Q* ensures that the number is in Q format, *m* is the two's complement integer portion of the number, and *n* is the number of bits used for the fractional part of the number.

The resolution of this Q format number is 2^{-n} , and its range lies in the interval $[-2^m; 2^m - 2^{-n}]$. To convert a Q format number into its floating-point equivalent, we use a power-of-two scaling. We thus multiply the Q format number by two to the power of the number of bits used for the fractional portion of the number:

$$B = A \times 2^n; \quad \text{where } B \text{ is the floating-point number and } A \text{ the Q format number} \quad (7.1)$$

The opposite operation (floating-point number to fixed-point number) is done by adding a negative sign in front of the number of bits used for the fractional portion of the number in Equation 7.1.

The reasoning behind each *CMSIS-NN* function optimization is explained in [14]. Note also that we use *CMSIS-NN* in this work due to the great benefits it yields, but other libraries begin to emerge nowadays. For instance, *CMIX-NN* [189] is a library allowing a neural network implementation using very low-precision fixed-point representation (8-4-2 bits). It is thus really interesting for low-bitwidth quantized neural networks. The Github repository can be found here [190]. Another library of interest is *PULP-NN* [191]. It is an optimized computing library for parallel ultra-low-power RISC-V processors targeting low-bitwidth quantized neural networks up to 1 bit (8-4-2-1 bit(s)). With the GAP-8 processor at its maximum clock frequency, *PULP-NN* claims to run $36.8\times$ faster than the *CMSIS-NN* Cortex-M4 reference implementation (with the STM32L4 MCU [14]) on the CIFAR-10 image classification task. In terms of energy efficiency, they say to outperform the *CMSIS-NN* Cortex-M4 reference implementation by a factor of 14.1 at the maximum efficiency operating point. The Github repository can be found here [192].

7.3 Conversion of a machine learning model into its *CMSIS-NN* implementation

Unfortunately, an automatic way to go from a *Keras* model to a *CMSIS-NN* implementation does not exist yet. However, thanks to the ARM company, it is possible to make the conversion from a *Caffe* model [193]. Nevertheless, there are a lot of works trying to solve this problem. For instance, [194] allows converting a *Keras* model to a *CMSIS-NN* implementation, but it supports, for the moment, only 1D convolution layers.

Due to the lack of conversion code, we need to handle the implementation manually in *CMSIS-NN*. We have to program it in a C language with the help of the *CMSIS-NN* functions. To achieve this task, we need to perform a lot of steps in a given order. The majority of these steps are based on [195]. We base our implementation on a post-training quantization, meaning that the network has already been trained before the conversion. However, it is possible to make the conversion in another way, referred to as the quantization-aware training. If it is the case, we do not need to consider the steps involving quantization. We can find such a framework here [196]. It allows training the network while considering the quantization noise and the saturation effects linked to the complex quantization process. However, "quantization-aware training is more complex and may not model all effects of the fixed-point implementation involving differences with the *CMSIS-NN* functions" [195]. It is the reason why we have chosen to use a post-training quantization in this work even though the accuracy loss is more significant compared to quantization-aware training. [197] proves for known complex architectures (MobileNet, Inception, ResNet) that the accuracy loss with 8-bit post-training quantization is on average of 2.2%. In contrast, the accuracy loss with 8-bit quantization-aware training is on average of 0.62%.

The remainder of this section is dedicated to the detailed explanation of each step to convert our network into its *CMSIS-NN* implementation.

7.3.1 Analyzing the *CMSIS-NN* functions

The first step is to check the compatibility between the network and the *CMSIS-NN* functions. The *CMSIS-NN* library is still under development by ARM and does not support all types of layers. If the layer is unavailable, we need to translate it into an equivalent combination of *CMSIS-NN* layers and *CMSIS-DSP* functions. If it is not possible, we cannot build the *CMSIS-NN* implementation. A good example, provided by the ARM tutorial [195], is the following: "There is no LSTM layer implementation in *CMSIS-NN*. But an LSTM layer can be expressed with *CMSIS-NN* and *CMSIS-DSP* using:

- Fully connected layers (*CMSIS-NN*) ;
- Sigmoid and hyperbolic tangent activations (*CMSIS-NN*) ;
- Elementwise vector product (*CMSIS-DSP*). "

We can see an overview of this step graphically in Figure 7.3.

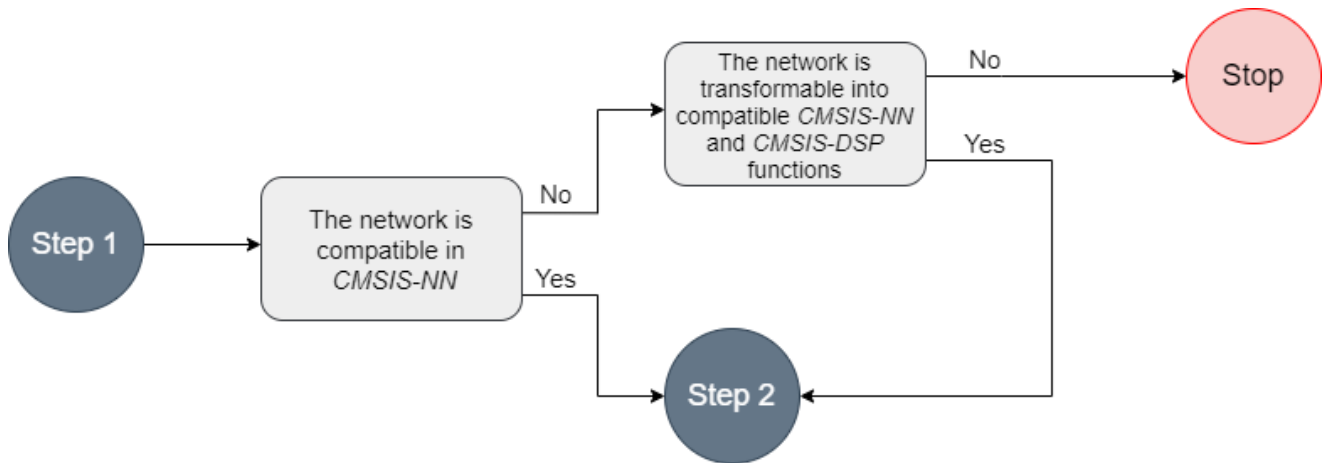


Figure 7.3: Overview of the step *Analyzing the CMSIS-NN functions*. **Step 1** is the starting point and **Step 2** the endpoint. **Stop** means that it is impossible to create the *CMSIS-NN* implementation of the network.

7.3.2 Analyzing the network

This step is optional and only applicable if the network contains depthwise separable convolutions. If it is not the case, we must bypass it.

The model built and trained in *Keras* uses **SeparableConv2D** layers. This layer is a depthwise separable convolution which consists of first a depthwise spatial convolution followed by a pointwise convolution. Therefore, the layer contains, in fact, two different operations, and we could separate it into two layers. It is the case in *CMSIS-NN*, for which we do not find a depthwise separable convolution function in itself. Instead, we find two functions, in different versions, doing the same job as a depthwise spatial convolution and a pointwise convolution. Such examples of *CMSIS-NN* functions are the *arm_depthwise_separable_conv_HWC_q7_nonsquare* and the *arm_convolve_1x1_HWC_q7_fast_nonsquare* functions. To access the distributions of activations at the output of the depthwise and pointwise convolutions, which eases step 7.3.6, it is a good practice to modify the back-end implementation. Therefore, we change the depthwise separable convolution

function by a depthwise spatial convolution function and a pointwise convolution function. Of course, we need to retrain the network, but no apparent modification should appear. The only difference is the absence of a bias set for the first operation in the **SeparableConv2D** layer. In contrast, two different functions add a bias set for each of those functions. The number of MACC operations increases thus a bit, but the accuracy remains the same.

At the end of this step, we thus replaced each depthwise separable convolution layer by a depthwise spatial convolution layer followed by a pointwise convolution layer. We show an overview of this step visually in Figure 7.4.

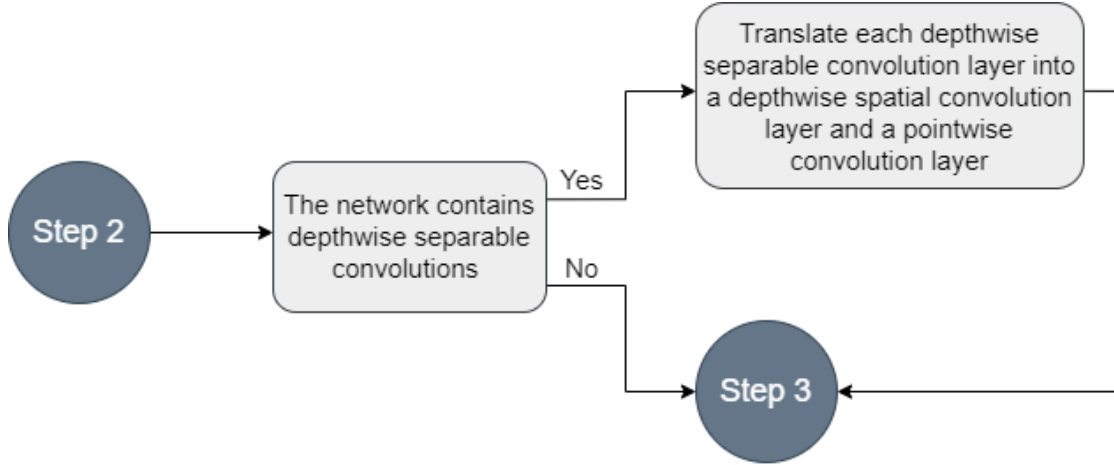


Figure 7.4: Overview of the step *Analyzing the network*. **Step 2** is the starting point and **Step 3** the endpoint.

7.3.3 Dealing with the batch normalization layers

This third step is only helpful if the network contains batch normalization layers. Otherwise, we must bypass it.

CMSIS-NN works with input and output buffers which are flattened arrays. Therefore, we do not find *CMSIS-NN* functions allowing us to perform a flatten layer as we could find in any machine learning framework (*Keras*, *PyTorch*, etc.). Furthermore, the dropout layer is not implemented in *CMSIS-NN* simply because the dropout is a regularization technique used during training to avoid overfitting, as explained in Section 2.5.2.3. However, the well-known batch normalization layer is also not implemented by the *CMSIS-NN* kernel. Because we use the batch normalization layer during the inference process, it needs to be part of the network. Fortunately, it is possible to bypass this problem by fusing the batch normalization layer with the previous layer. The methodology to modify the weights and biases of a regular convolution layer to incorporate the batch normalization layer effect is the following:

$$W = \gamma \times \frac{W_{conv}}{\sqrt{\sigma^2 + \epsilon}}; \quad (7.2)$$

$$b = \gamma \times \frac{b_{conv} - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta; \quad (7.3)$$

where W represents the new convolution weights, W_{conv} represents the initial convolution weights, b is the new convolution bias, b_{conv} is the initial convolution bias and γ , μ , σ^2 and β are the batch

normalization parameters as explained in Section 2.5.2.4. The smoothing term (ϵ) is the same parameter used in the batch normalization layers, a small float value typically equal to 0.001, added to the variance to avoid dividing by zero. Note that its use is primordial since "small differences in each activation can quickly create butterfly effects in the output" [198].

The explanation behind these two equations is fully detailed in [198]. This article also explains how to fuse batch normalization weights in other layers such as a dense layer, a depthwise convolution, a transpose convolution, and a depthwise separable convolution. The latter interests us since the network contains this kind of layer. For the depthwise separable convolution, translated because of step 7.3.2 into a depthwise spatial convolution and a pointwise convolution, we need to apply Equations 7.2 and 7.3 to the pointwise convolution only. Note that the batch normalization folding with the previous layer is possible for a convolution layer followed by batch normalization and the activation layer, as it is the case in our network. Suppose the activation layer is before the batch normalization layer, which can arrive because it yields better results according to some benchmarks [199]. In that case, we cannot fuse the batch normalization layer with the previous layer because of the ReLU non-linearity. However, it should be possible to merge the batch normalization weights into the weights of the next convolution layer following Equations 7.4 and 7.5 if we do not use padding. Otherwise, it is more complicated to handle as mentioned in [198].

$$W = \gamma \times \frac{W_{conv}}{\sqrt{\sigma^2 + \epsilon}}; \quad (7.4)$$

$$b = W_{conv} \times \gamma \times \frac{\beta - \mu}{\sqrt{\sigma^2 + \epsilon}} + b_{conv}; \quad (7.5)$$

At the end of this step, the batch normalization layers are removed from the architecture, as their weights are embedded into the convolution layers. We present an overview of this step in Figure 7.5.

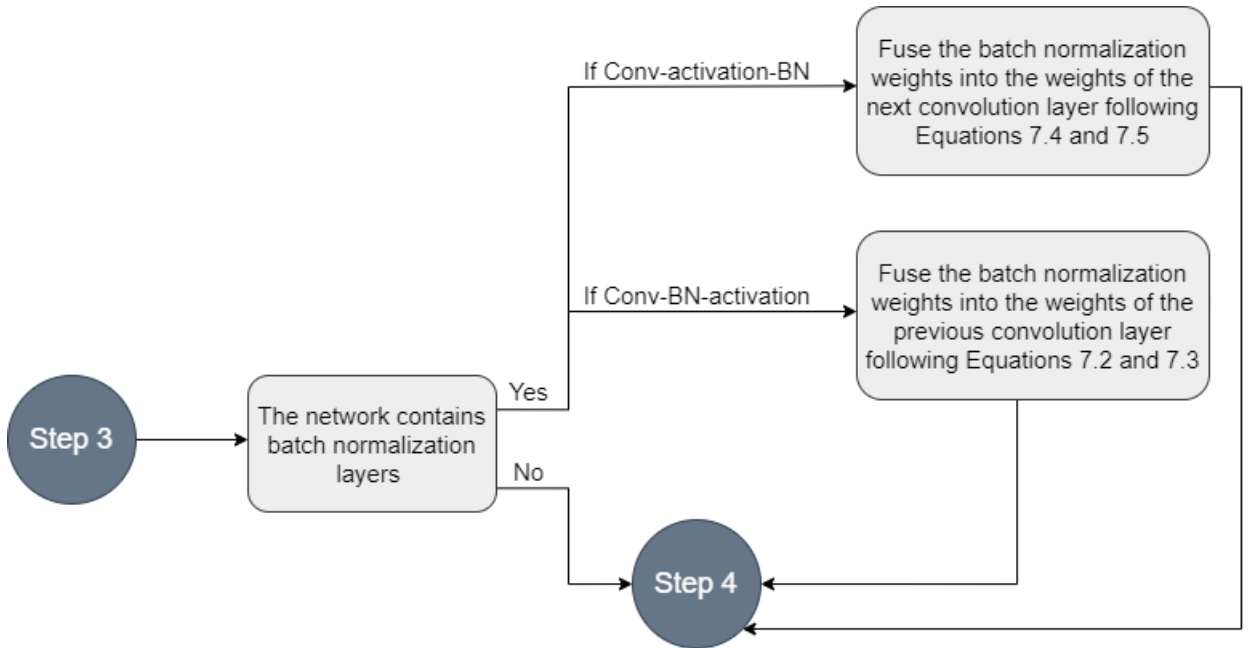


Figure 7.5: Overview of the step *Dealing with the batch normalization layers*. **Step 3** is the starting point and **Step 4** the endpoint.

7.3.4 Reordering of the weights

We perform this step on all the layers containing weights. For instance, pooling or activation layers are not handled in this step.

The *CMSIS-NN* functions expect a particular ordering of the weights for each layer. On the one hand, convolution weights are expected to be in the **NHWC** format where **N** denotes the number of output channels, **H** is the height of the input, **W** is the width of the input, and **C** is the number of input channels. This ordering is simply for optimization purposes where ARM proved in [14] that the **HWC** data layout compared to **CHW** led to a reduced runtime on their implementation of the convolution operation. On the other hand, fully connected weights are expected to be in the **NC** format. In practice, depending on the framework used for the implementation and training of the network, it is not always the case. For example, in the *Keras* framework, convolution weights are obtained in the **HWCN** format, while fully connected weights are in the **CN** format.

The following piece of Python code allows doing the weight reordering for a convolution layer and a fully connected layer. Note that we use a specific and more complicated reordering of the fully connected weights to use the most optimized fully connected *CMSIS-NN* function (*arm_fully_connected_q7_opt*). We can find the *convert_to_x4_q7_weights* function used for this purpose in [200]. ARM created the function and released it to be simple to use. If the performance of the fully connected layer is not of interest to the application, we can remove the third line in the piece of code below.

```
1 Reordered_Conv_weights = (np.transpose(Conv_Weights, (3, 0, 1, 2))).flatten()
2 wts = np.transpose(FullyConnected_Weights)
3 reordered_wts = convert_to_x4_q7_weights(np.reshape(wts, (wts.shape[0], wts.shape[1], 1, 1)))
```

We do not reorder the biases since they contain only one dimension, equal to the number of output channels. Furthermore, a flattening is used at the end because *CMSIS-NN* only works with flattened arrays (for the fully connected layer, the *convert_to_x4_q7_weights* function makes a flattening of the weights internally).

Depending on the framework we are working with, and thus on the format of the weights obtained by the latter, we should modify the arguments of the *transpose* function accordingly. For instance, if the structure of the convolution weights from the framework is **HWNC**, and if we want to obtain the weights in the **NHWC** format, we should transpose the weights with the arguments: (2, 0, 1, 3). The same reasoning is applicable for the fully connected weights.

At the end of this step, the weights of the convolution and fully connected layers are ordered correctly for *CMSIS-NN* and flattened. We display an overview of this step in Figure 7.6.

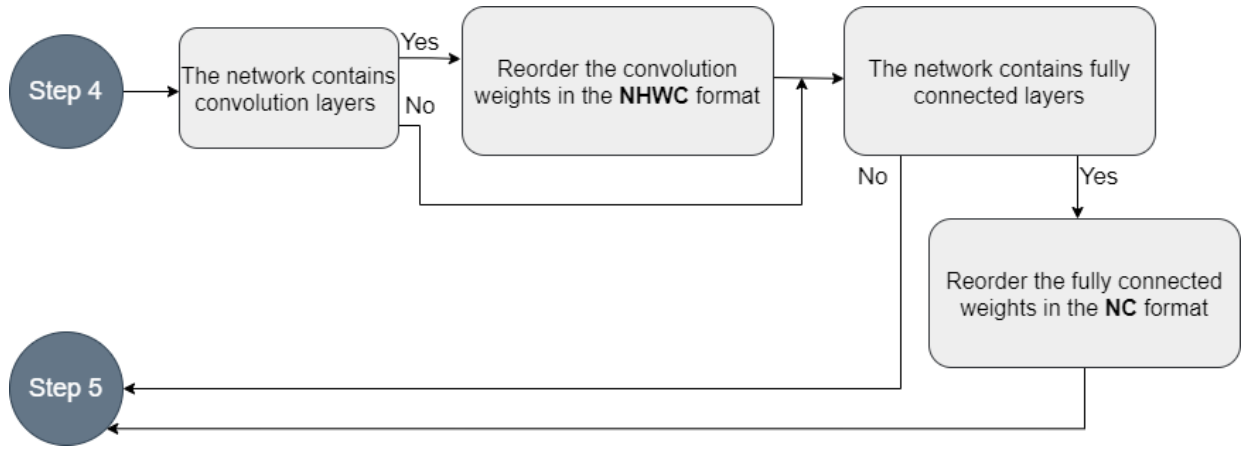


Figure 7.6: Overview of the step *Reordering of the weights*. **Step 4** is the starting point and **Step 5** the endpoint.

7.3.5 Quantization of the weights

We perform this step on all the layers that contain weights. For instance, pooling or activation layers are not handled in this step. As explained in Section 7.2, *CMSIS-NN* only works with 8-bit or 16-bit data. The weights and biases, initially in floating-point format (32-bit), need to be quantized to a lower precision. In this work, we chose an 8-bit post-training quantization since it yields the smallest memory requirements (factor of 4) while limiting the accuracy loss to approximately 2.2% [197], which is acceptable in this application. With 8 bits, we have eight possible Q formats going from Q7.0 to Q0.7. A Q format is the same for all the weight values belonging to the same layer, but the Q format of the bias of the given layer is not required to be the same. Therefore, to choose the best Q format for the weights and biases of a given layer, we use a quantization scheme that takes the Q format minimizing the approximation error (the error between the quantized values and the initial ones). We give a pseudo-code of the quantization scheme below. The second and third algorithms are the functions *quantize* and *dequantize* used in the first one. In reality, the *quantize* function is simply the algorithm used to go from a floating-point number to a Q format number. In contrast, the *dequantize* function does the opposite operation.

Algorithm 1 8-bit quantization scheme : first-order error minimisation quantization

Input: *Array*

Output: *Quantized_array*

```

1: Possible_scales =  $[2^{-7}, 2^{-6}, 2^{-5}, 2^{-4}, 2^{-3}, 2^{-2}, 2^{-1}]$ 
2: Tested_scale =  $2^0$ 
3: min_diff =  $\sum_i |Array[i] - dequantize(quantize(Array, Tested\_scale), Tested\_scale)[i]|$ 
4: for scale in Possible_scales do
5:   quantized = quantize(Array, scale)
6:   if  $\max(quantized) \leq int8\_max\_range$  AND  $\min(quantized) \geq int8\_min\_range$  then
7:     diff =  $\sum_i |Array[i] - dequantize(quantized, scale)[i]|$ 
8:     if diff < min_diff then
9:       min_diff = diff
10:      Scale_quant = scale
11:    end if
12:  end if
13: end for
14: Quantized_array = int8(quantize(Array, Scale_quant))
  
```

Algorithm 2 *quantize* function

Input: *Array, Scale* : *Scale* is part of the array "*Possible_scales*" from the first algorithm

Output: *Quantized_array*

1: $Quantized_array = round(Array \times Scale)$

Algorithm 3 *dequantize* function

Input: *Quantized_array, Scale* : *Scale* is part of the array "*Possible_scales*" from the first algorithm

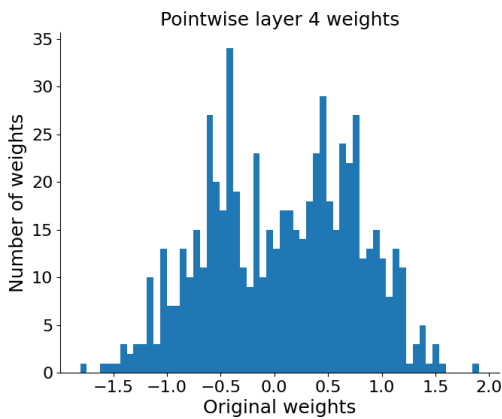
Output: *Array*

1: $Array = Quantized_array / Scale$

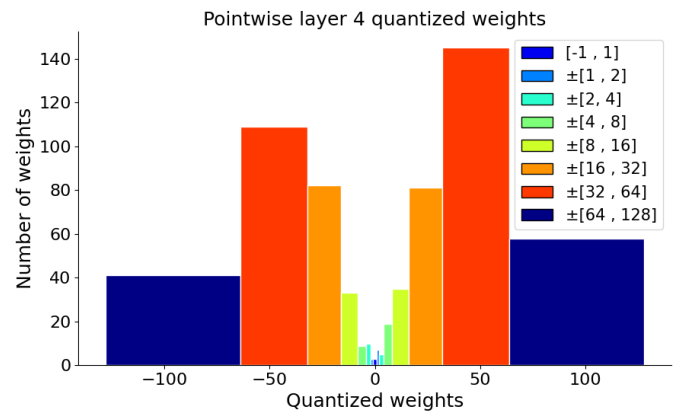
After applying the presented quantization scheme on the weights and biases of each layer of the network obtained at the end of step 7.3.4, we get the Q formats represented in Table 7.1. Then, the quantized and flattened weights of each trainable layer need to be exported to a header file on the local computer. For instance, the weights of the last pointwise convolution layer are exported to the file *LastPointwiseConvolutionWeights.h* while the bias set of this layer is exported to the file *LastPointwiseConvolutionBias.h*. Figure 7.7 shows, as an example, the distribution of weights of the last pointwise convolution layer. We present the weights from the initial floating-point network on the left of the figure, and the quantized weights with the Q1.6 format are shown on the right.

Layer	Weights Q format	Biases Q format
Depthwise convolution 1	Q0.7	Q0.7
Pointwise convolution 1	Q3.4	Q2.5
Depthwise convolution 2	Q0.7	Q0.7
Pointwise convolution 2	Q1.6	Q3.4
Depthwise convolution 3	Q0.7	Q0.7
Pointwise convolution 3	Q1.6	Q2.5
Depthwise convolution 4	Q0.7	Q0.7
Pointwise convolution 4	Q1.6	Q3.4
Fully connected	Q0.7	Q0.7

Table 7.1: Q formats obtained for the weights and biases of each layer of the network.



(a) Floating-point weights from the initial network.



(b) Fixed-point weights from the quantized network.

Figure 7.7: Original and quantized weights of the last pointwise convolution layer of our network.

At the end of this step, we quantized the weights and biases of each trainable layer. We also exported these quantized weights and biases in different header files and saved all the Q formats for each trainable layer because they are reused in step 7.3.7. An overview of this step is given in Figure 7.8.

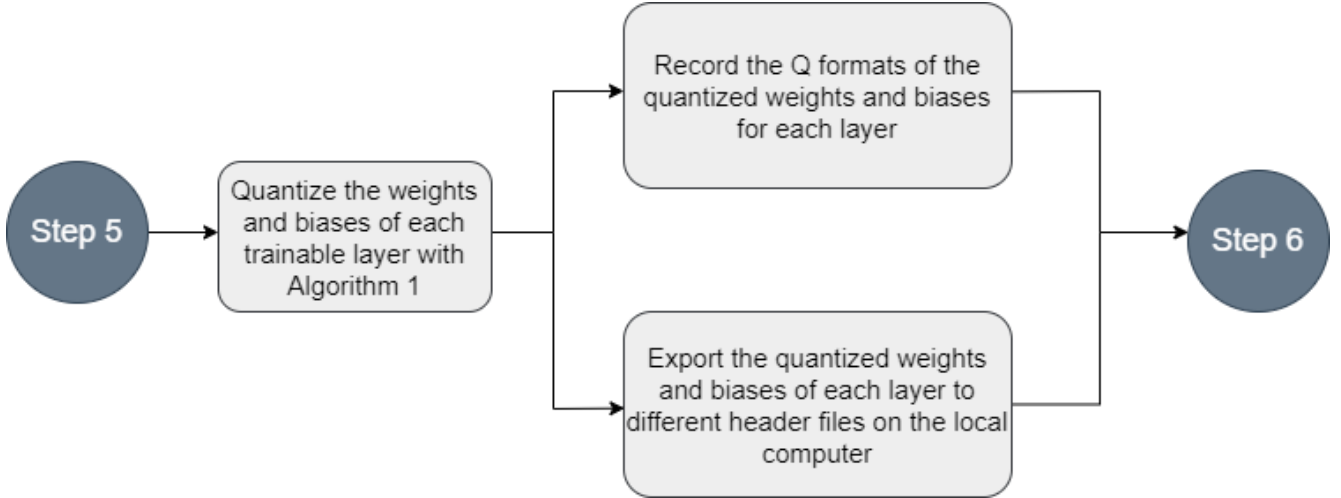


Figure 7.8: Overview of the step *Quantization of the weights*. **Step 5** is the starting point and **Step 6** the endpoint.

7.3.6 Quantization of the activations

In this context, the activation is the data at the input or the output of a layer. It must not be confused with an activation layer. Because *CMSIS-NN* works only with fixed-point data, we must also quantize the activations with 8 bits. We thus want to find the Q formats for the input and output of each layer. However, it is not as simple as for the weights quantization since we do not know the range of values in a specific layer. A trick to perform the activation quantization is to take the whole training set (or validation set), infer it through the trained floating-point network, and measure the statistical distribution of the activation data of each layer. At the end of the process, we can thus have a histogram per output layer showing the distribution of the data. Such an example is displayed in Fig. 7.9 where the output of the last depthwise convolution layer, the output of the last pointwise convolution layer, and the output of the fully connected layer histograms are shown. Note that they have been obtained by inferring the whole validation set through the trained floating-point network. With this information, multiple quantization schemes can be used and tested to find the one giving the best results. It is important to try various schemes since it can make a big difference depending on the network architecture and the dataset. We choose to test the **max-min quantization** and the **max-min quantization without outliers** among all possible quantization schemes. The first scheme uses the full range of values found in each histogram. It does not take into account the distribution of the data. The second quantization scheme is based on the same idea but allows the removal of a percentage of outliers. It thus uses the distribution of the data. We decide to test two configurations: **98% max-min quantization without outliers** and **90% max-min quantization without outliers**. In the first configuration, we remove 1% on the left and the right of the histograms, whereas the second configuration removes 5% on both sides. Note that the two chosen quantization schemes are the simplest ones. There exists a lot of interesting quantization schemes to obtain a better accuracy: the *Kullback-Leibler Divergence (KLD)* quantization [201] and the *Analytical Clipping for Integer Quantization (ACIQ)* [202] are two known examples.

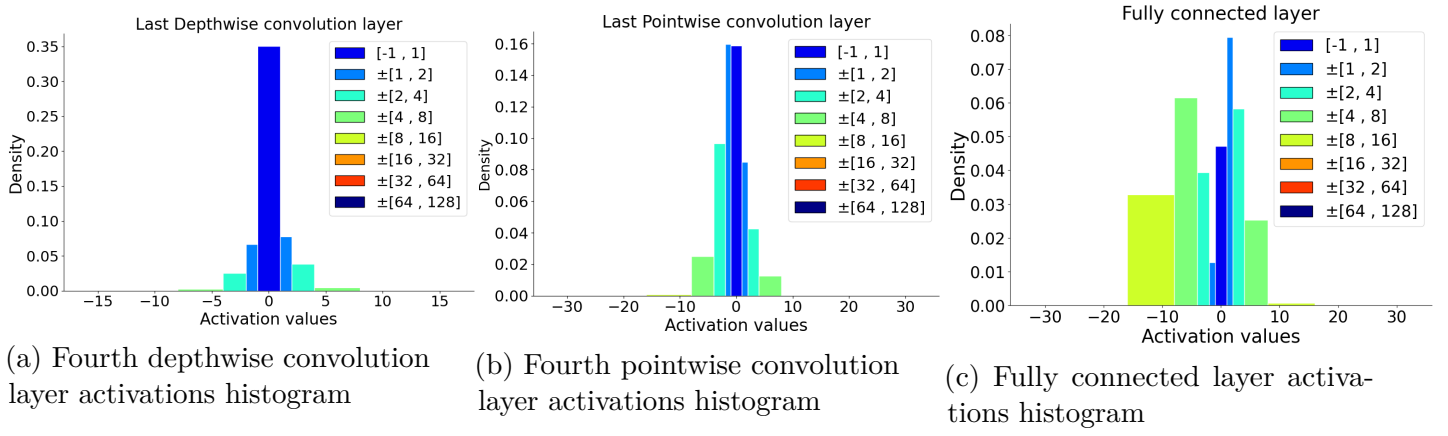


Figure 7.9: Activations histograms of three different layers. The activations are floating-point values. The histograms bins scale at a power-of-two as it allows knowing the possible Q formats.

With the histogram of each layer and the chosen quantization schemes, we can find the Q formats for the input and output of the layers. It is essential to mention that only some layers involve a possible modification of the Q format. These layers are the convolution layers (and their siblings) and the fully connected layers. For instance, a ReLU activation or a pooling layer requires the output Q format to be the same as the input Q format. It is due to the *CMSIS-NN* implementation. It is thus only helpful to record the histograms of the convolution and fully connected layers before their activation layers. For instance, we are interested in the output format of the final fully connected (dense) layer and not the output format of the softmax layer since we know it will be the same as the previous dense layer. A complete schematic to find the Q formats of all layers is shown in Figure 7.10.

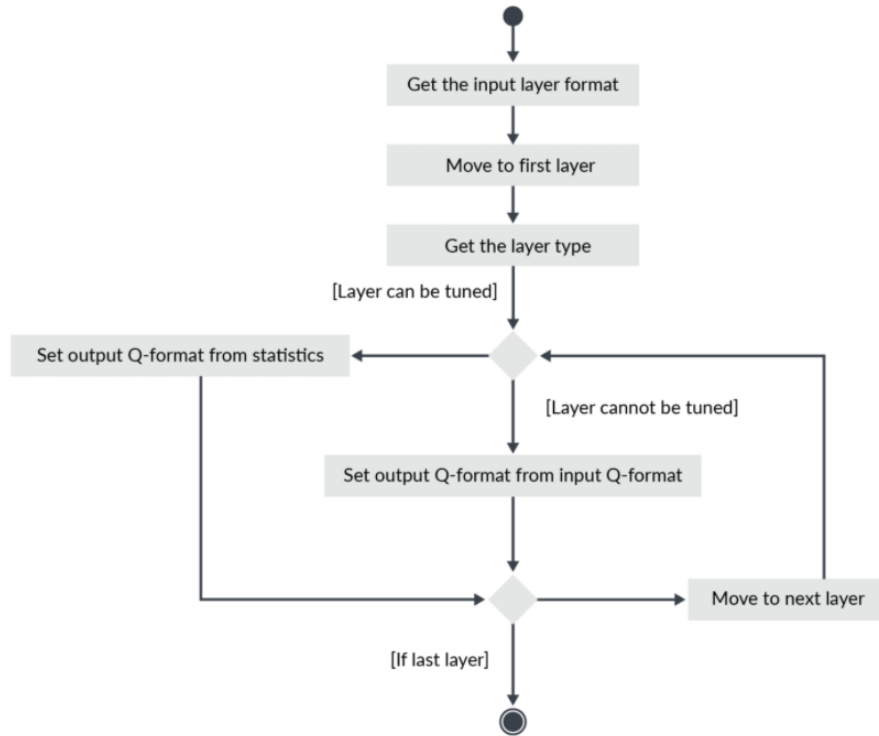


Figure 7.10: Pipeline to find the input and output Q formats of each layer in the *CMSIS-NN* implementation [195].

By applying this pipeline on our network with the chosen quantization schemes, we obtain these Q formats for each layer in Table 7.2.

Layer/Scheme	Max-Min Valida- tion	Max-Min Train	Max-Min w/o outliers 98% Vali- dation	Max-Min w/o outliers 98% Train	Max-Min w/o outliers 90% Vali- dation	Max-Min w/o outliers 90% Train
Input	Q4.3	Q4.3	Q3.4	Q3.4	Q0.7	Q0.7
Depthwise convolution 1	Q4.3	Q4.3	Q2.5	Q2.5	Q1.6	Q1.6
Pointwise convolution 1	Q6.1	Q7.0	Q4.3	Q4.3	Q2.5	Q2.5
Depthwise convolution 2	Q5.2	Q6.1	Q4.3	Q4.3	Q3.4	Q3.4
Pointwise convolution 2	Q6.1	Q6.1	Q4.3	Q4.3	Q3.4	Q3.4
Depthwise convolution 3	Q5.2	Q6.1	Q4.3	Q4.3	Q3.4	Q3.4
Pointwise convolution 3	Q6.1	Q6.1	Q4.3	Q4.3	Q3.4	Q3.4
Depthwise convolution 4	Q5.2	Q5.2	Q3.4	Q3.4	Q2.5	Q2.5
Pointwise convolution 4	Q6.1	Q6.1	Q3.4	Q3.4	Q3.4	Q3.4
Fully connected	Q5.2	Q5.2	Q4.3	Q4.3	Q4.3	Q4.3

Table 7.2: Q formats obtained for the activation values at the **output** of each layer for each quantization scheme. The input Q format of a layer is the output Q format of the previous layer. The activation layers are not shown in this table since their output Q format must be the same as their input Q format, meaning that it must be the same as the output Q format of the previous layer. The keyword **Validation** means that the whole validation set has been used to obtain the histograms, while **Train** refers to the entire training set.

At the end of this step, we obtained the Q formats at the output of each layer. We saved these values to be reused in the next step. We present an overview of this step in Figure 7.11.

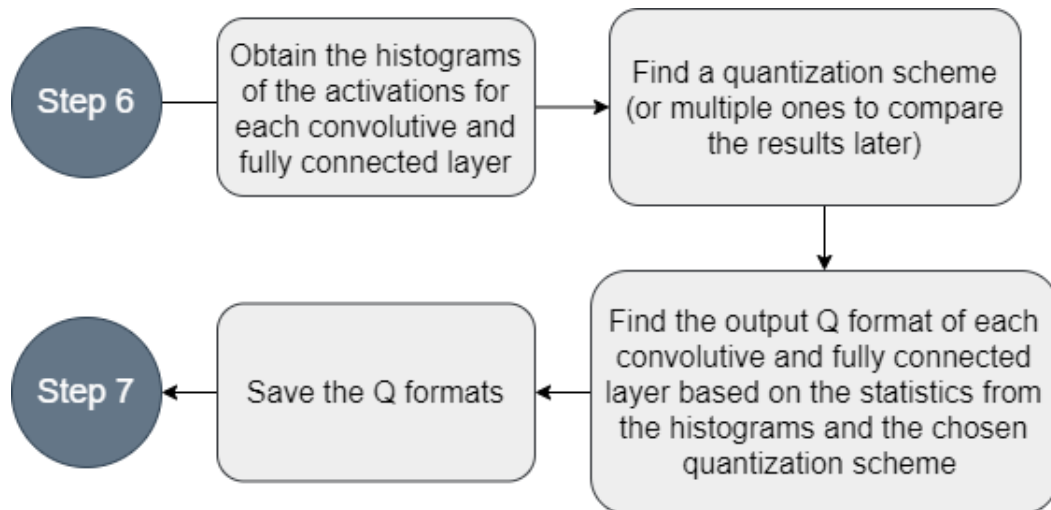


Figure 7.11: Overview of the step *Quantization of the activations*. **Step 6** is the starting point and **Step 7** the endpoint.

7.3.7 Computation of the bias and output shifts

When the weights, biases, and activation data have been quantized, we know their Q formats. However, we need to discuss the bias and output shifts that convolution and fully connected *CMSIS-NN* functions take as arguments. Only these two types of layers use the bias and output shifts arguments because they are the only ones for which the output Q format can change with the input Q format, as explained in step 7.3.6. This system of shifts is essential for *CMSIS-NN* since all values are represented as 8-bit integers. Therefore, adjusting the difference between quantized values that do not have the same Q format is needed. In other words, the shift system ensures that the output Q format is the same as the one requested by the layer. Let us take a simple example released by ARM [203] that allows understanding it : "For example, from quantization sweeps we get the input to the first fully-connected layer in DNN model is of Q5.2 format or 2 bits after the decimal point (i.e. ranges from -32, 31.75) and the expected output format for maximum accuracy is also Q5.2. Let us assume that the quantized weights are of the format Q0.7 (i.e., in range -1, 1) and the quantized biases have 8 bits for fractional point (i.e., range -0.5, 0.5). So, the product (layer_input $\times$ weights) will have 9 bits in the fractional part (Q5.2 $\times$ Q0.7 = Qx.9), and biases need to be shifted left by 1 to get to the same representation (i.e., Qx.8 < 1 = Qy.9). The layer output has 2 bits for fractional part (Q5.2), so the product (layer_input $\times$ weights) needs to be shifted right by 7 to get to Q5.2 format."

The simple equations allowing to find the bias shift and output shift of each layer (convolution and fully connected) are given here and come from [195] :

$$\text{Bias shift} = (f_i + f_w) - f_b; \quad (7.6)$$

$$\text{Output shift} = (f_i + f_w) - f_o; \quad (7.7)$$

where f_i , f_o , f_w , and f_b are respectively the number of fractional bits for the input, output, weights, and bias.

These shifts can be calculated using Table 7.1 and Table 7.2. The bias shifts obtained for our network, for the three quantization schemes, are represented in Table 7.3, whereas the output shifts are shown in Table 7.4.

Layer/Scheme	Max-Min Valida- tion	Max-Min Train	Max-Min w/o outliers 98% Vali- dation	Max-Min w/o outliers 98% Train	Max-Min w/o outliers 90% Vali- dation	Max-Min w/o outliers 90% Train
Depthwise convolution 1	3	3	4	4	7	7
Pointwise convolution 1	2	2	4	4	5	5
Depthwise convolution 2	1	0	3	3	5	5
Pointwise convolution 2	4	3	5	5	6	6
Depthwise convolution 3	1	1	3	3	4	4
Pointwise convolution 3	3	2	4	4	5	5
Depthwise convolution 4	1	1	3	3	4	4
Pointwise convolution 4	4	4	6	6	7	7
Fully connected	1	1	4	4	4	4

Table 7.3: Bias shifts obtained for each layer and each quantization scheme. The keyword **Validation** means that the whole validation set has been used to obtain the histograms, while **Train** refers to the entire training set.

Layer/Scheme	Max-Min Valida- tion	Max-Min Train	Max-Min w/o outliers 98% Vali- dation	Max-Min w/o outliers 98% Train	Max-Min w/o outliers 90% Vali- dation	Max-Min w/o outliers 90% Train
Depthwise convolution 1	7	7	6	6	8	8
Pointwise convolution 1	6	7	6	6	5	5
Depthwise convolution 2	6	6	7	7	8	8
Pointwise convolution 2	7	6	6	6	6	6
Depthwise convolution 3	6	7	7	7	7	7
Pointwise convolution 3	7	6	6	6	6	6
Depthwise convolution 4	6	6	6	6	6	6
Pointwise convolution 4	7	7	6	6	7	7
Fully connected	6	6	8	8	8	8

Table 7.4: Output shifts obtained for each layer and each quantization scheme. The keyword **Validation** means that the whole validation set has been used to obtain the histograms, while **Train** refers to the entire training set.

At the end of this step, we obtained the bias and output shifts needed for the convolution and fully connected functions in *CMSIS-NN*. These values are saved for reuse in the final implementation in the next step. An overview of this step is presented in Figure 7.12.

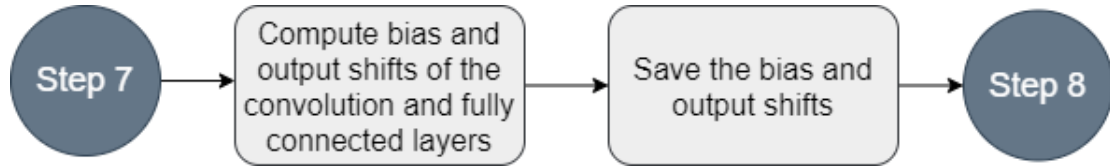


Figure 7.12: Overview of the step *Computation of the bias and output shifts*. **Step 7** is the starting point and **Step 8** the end point.

7.3.8 Generation of the *CMSIS-NN* implementation

When the Q formats for the weights, biases, and activations are known as well as the bias and output shifts for each layer, we can generate the final *CMSIS-NN* implementation in C language. For good practice reasons, we use different files to have a clear overview of the code. Figure 7.13 shows the disposition of the files.

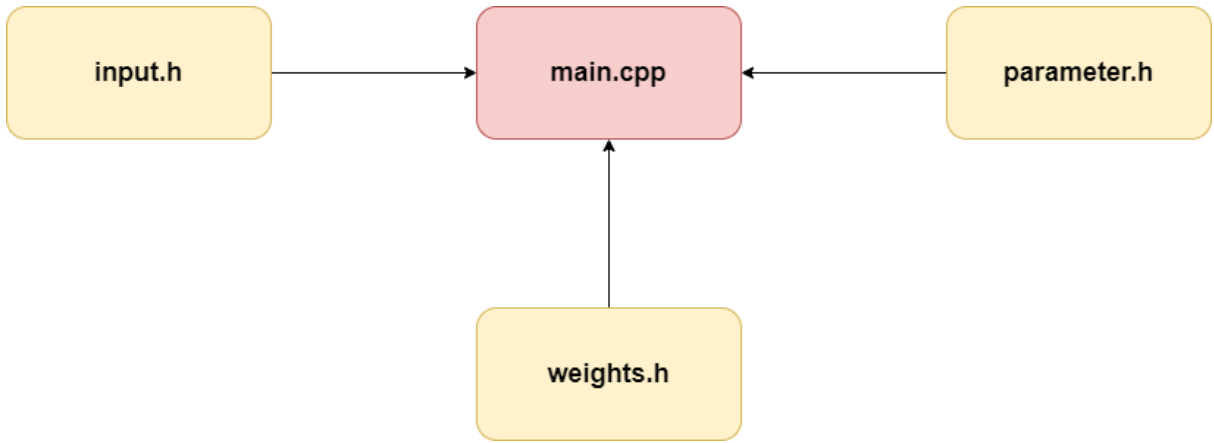


Figure 7.13: The different files used in the final *CMSIS-NN* implementation. The file in red is the most important one. It contains the application and the convolutional neural network implementation. The files in yellow are secondary files that are used by the main file.

The first file (**main.cpp**) is the main file. It contains the *CMSIS-NN* implementation with the use of *CMSIS-NN* functions. Each layer in the initial framework is translated into a corresponding *CMSIS-NN* function. For instance, a fully connected layer with 8-bit data types is interpreted either by the `arm_fully_connected_q7` function or the `arm_fully_connected_q7_opt` function. The ReLU activation with 8-bit data types is transcribed by the `arm_relu_q7` function. The parameters of each function are passed to the corresponding function through the use of buffers. These buffers are initialized before the call to the main function with the correct size depending on the parameters. They contain values coming from the **parameter.h** file, the **weights.h** file and the **input.h** file. For instance, the buffer used to store the ten weights of the first depthwise convolution has 10 bytes. These files contain the parameter values, the weight values, and the input MFCC feature already preprocessed, as explained further in this section. Furthermore, two other buffers, known in the literature as the **scratch_buffer** and the **col_buffer** are needed for an efficient implementation. The first one contains the activation data (input and output data of each layer), while the second is used for the *im2col* output (used by the convolution functions from *CMSIS-NN*). The **scratch_buffer** is divided into two more buffers named **img_buffer1** and **img_buffer2**. These two contain one after the other the input data or the output data of each layer. Their size depends on the network architecture. The use of only two buffers used one after the other, as input-output buffers, is a trick to save RAM, which has been implemented by ARM [204]. The idea is to overwrite the data of the buffers which are not used.

The methodology to calculate the size of the **scratch_buffer** is represented in Fig. 7.14 as well as the switch between layers for **img_buffer1** and **img_buffer2**. Also note that activation layers do not switch between buffers since they simply overwrite the output from the previous layer. For this reason, they are not represented in Figure 7.14. For each of the two buffers, the idea consists of finding the layer that requires the maximum buffer size to store the intermediate layer inputs-outputs. Then, the **scratch_buffer** size is simply obtained by joining the two maximal sizes. Also note that we use a final buffer, referred to as **output_data**. It contains the final softmax results for each class from the network inference. For the **col_buffer** size, we need to compute the following formula, on the biggest memory-consuming layer, that predicts the required size and comes from [182] :

$$\text{col_buffer size} = 2 \times 2 \times (\text{kernel width}) \times (\text{kernel height}) \times (\text{number of filters}) \quad (7.8)$$

In our network, the maximal size is obtained with 3x3 kernels and 40 output feature maps that therefore involve a **col_buffer** size of 1440 bytes. By combining the length of **col_buffer** and

scratch_buffer, we need 3040 bytes of RAM, which is extremely small compared to tinyCNNs from state-of-the-art, such as the CIFAR-10 CNN released by ARM [205] with 43.1 kB of RAM usage. Moreover, this RAM usage is well below the total RAM size of the microcontroller: approximately 0.79% of the total available RAM).

The second file (**parameter.h**) contains all the parameters of each *CMSIS-NN* function. We can find the parameters of each *CMSIS-NN* function in the *CMSIS-NN* libraries [185], and they depend on the architecture of the neural network. For instance, the parameter *ch_im_in* is the number of input channels, while the parameter *dim_im_in_x* is the input tensor dimension along the x dimension. However, a piece of advice would be to be careful about each parameter of each function. Simple programming mistakes can happen easily because of a misunderstanding of a parameter. For instance, the *dim_im_in_x* parameter does not want the height of the input tensor but the width (it thus expects the second dimension of the input tensor).

The third file (**weights.h**) contains, for each trainable layer, the weights and the bias set associated with the layer as well as the corresponding bias shift and output shift. If we carefully followed the previous steps, all these values should be stored in the local computer, and the remaining step is only to put these in the **weights.h** file.

Finally, the last file (**input.h**) is optional and depends on the application. In this case, the file contains a test data sample already preprocessed into an MFCC feature. It is thus an image that has been flattened (**line after line** and not column after column) and that is inferred through the network. In a final implementation, this file should not exist: the feature would be preprocessed directly on the board or streamed from another device. Here, this file is for debugging and testing purposes.

All the files used to make the *CMSIS-NN* implementation are shown in Appendix A.5. Note that a **max-min quantization without outliers at 98%** scheme has been used for the activations quantization in this example.

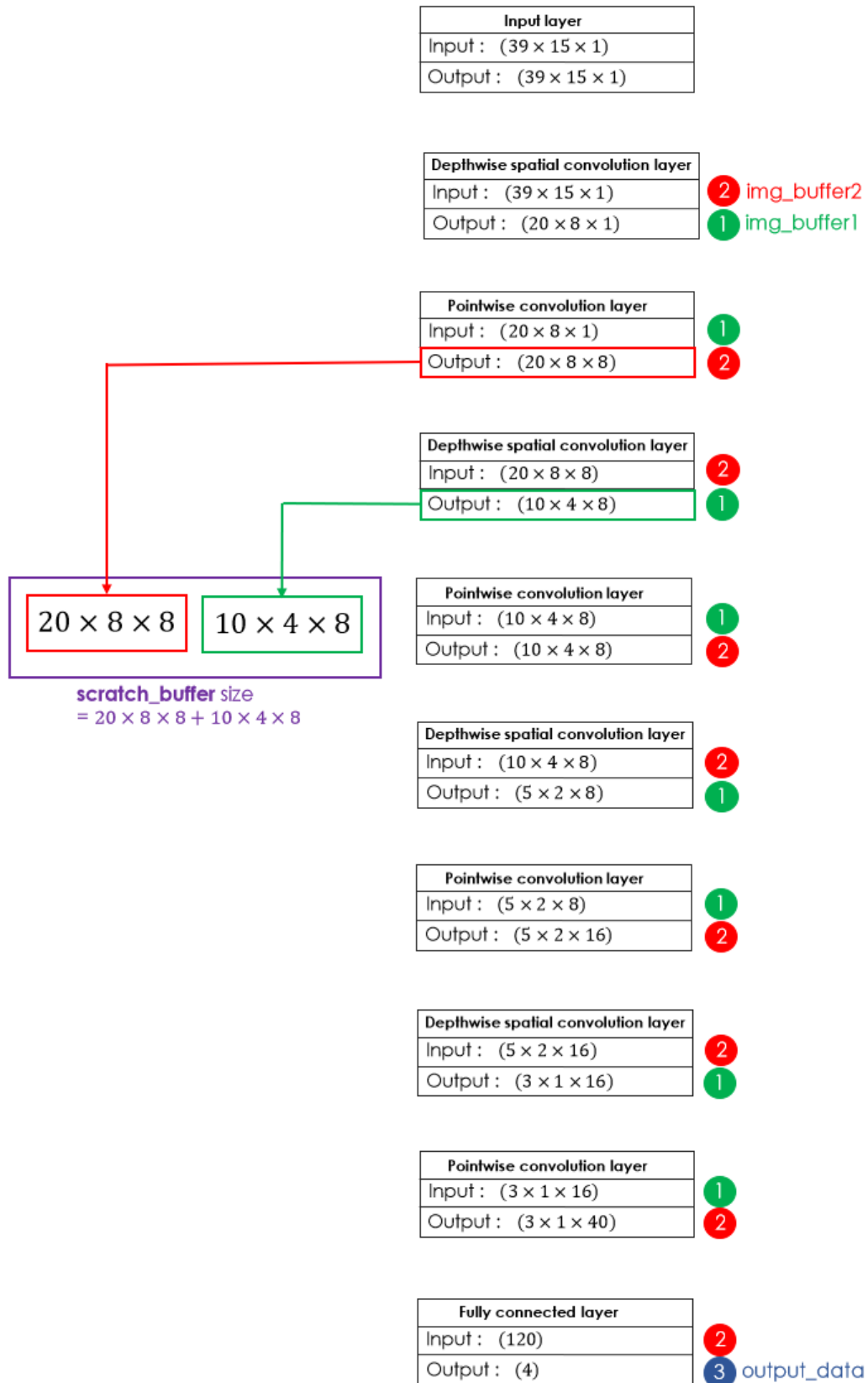


Figure 7.14: Methodology to calculate the size of the **scratch_buffer**. Its total size is of 1600 bytes.

An overview of the final step is shown in Figure 7.15.

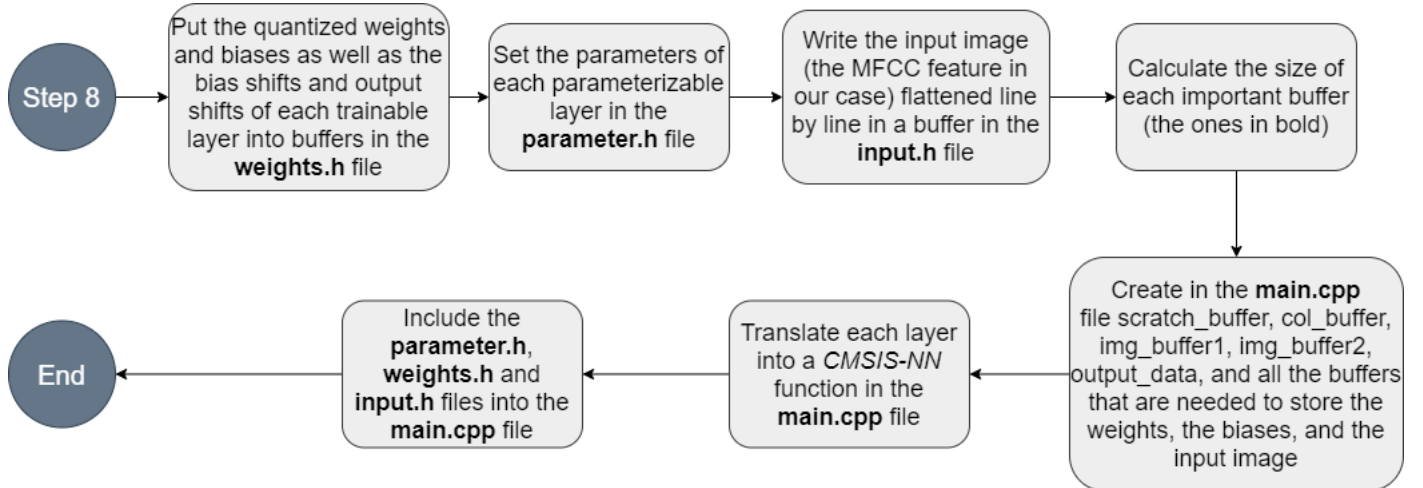


Figure 7.15: Overview of the step *Generation of the CMSIS-NN implementation*. **Step 8** is the starting point and **End** the finalization of the complete process to convert a network into its *CMSIS-NN* implementation.

7.4 Metrics of the network with *CMSIS-NN*

Because the convolutional neural network is now embedded into the Apollo3 Blue microcontroller, we can obtain important metrics out of it : the test set accuracy, the runtime execution of the network, and the power consumption of the device. Let us analyze each metric and explain how they are computed.

7.4.1 Test set accuracy

Because of the quantization process and the fixed-point behavior of *CMSIS-NN*, an accuracy loss on the test set is expected compared to the refined floating-point network. To have the answer, two possibilities can be used to infer the test set through the *CMSIS-NN* network. Note that the test set has to be pre-processed beforehand into MFCC features since we do not focus on the feature pre-processing step in this work.

- Using the UART communication. The idea is to send, from the local computer to the microcontroller, pre-processed samples from the test set (a MFCC feature) one by one until all test samples have been sent. For each image received by the microcontroller, the inference is made, and the result (the predicted label) is sent this time from the microcontroller to the local computer. At the end of the process, the local computer would contain all the true labels and the predicted labels stored into arrays, and we could obtain the test set accuracy based on them. The utopic way would be to put in the ROM of the microcontroller all the pre-processed test samples and to perform the inferences to obtain the test accuracy. However, the ROM is too small for that purpose ;
- Using the simulator from *uVision5*. This technique does not use the microcontroller but simulates the behavior of the microcontroller in software. The advantage is that the amount of memory is tremendous (up to 0x99999999 bytes in hexadecimal), meaning that we can put in the **input.h** file of the *CMSIS-NN* implementation the entire pre-processed test set (flattened line by line). The inference can then be simulated, still with the *CMSIS-NN* functions.

Because the first technique is time-consuming due to its implementation and its complexity, we have decided to use the second one. Therefore, we do not use for this experiment the microcontroller, but we simulate it. However, it is acceptable since the purpose is to compare the test accuracy between the floating-point network and the fixed-point network with the *CMSIS-NN functions*, and these functions are used with this technique.

Multiple embedded networks have been tested due to the different activations quantization schemes from Section 7.3.6. The results are shown in Table 7.5.

	Max-Min Valida- tion	Max-Min Train	Max-Min w/o outliers 98% Vali- dation	Max-Min w/o outliers 98% Train	Max-Min w/o outliers 90% Vali- dation	Max-Min w/o outliers 90% Train	Refined floating- point network
Test accuracy	0.885	0.894	0.914	0.914	0.494	0.494	0.935

Table 7.5: Test set accuracy of the *CMSIS-NN* network depending on the activations quantization scheme as well as the test set accuracy of the refined floating-point network.

A 2.1% accuracy loss can be observed with the best activations quantization scheme (**Max-Min without outliers at 98%**). This is close to the expected loss (2.2% from [197]) mentioned in Section 7.3. The confusion matrixes of the *CMSIS-NN* network with the best activations quantization scheme and the refined floating-point network are analyzed in Figure 7.16. Moreover, the precision, the recall and the F1-score of each class for both networks are presented in Table 7.6.

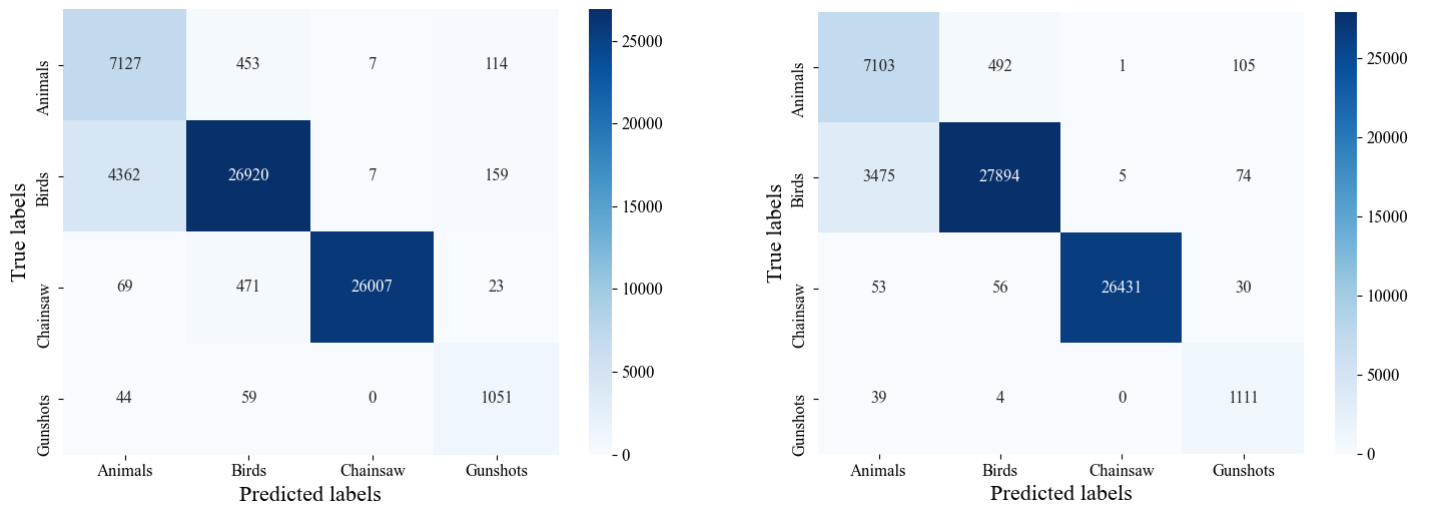


Figure 7.16: Confusion matrix on the test set.

Metric	Class	<i>CMSIS-NN</i> network	Floating-point network
Precision	Animals	0.614	0.666
	Birds	0.965	0.981
	Chainsaws	0.999	1.0
	Gunshots	0.78	0.842
	Weighted average	0.935	0.95
Recall	Animals	0.925	0.922
	Birds	0.856	0.887
	Chainsaws	0.979	0.995
	Gunshots	0.911	0.963
	Weighted average	0.914	0.935
F1-score	Animals	0.738	0.773
	Birds	0.907	0.931
	Chainsaws	0.989	0.997
	Gunshots	0.84	0.898
	Weighted average	0.919	0.939

Table 7.6: Metrics analysis of the *CMSIS-NN* network compared to the refined floating-point network in terms of precision, recall and F1-score.

We can observe in Table 7.6 that each metric is degraded between 1.5 and 2% for the entire dataset due to the quantization process. More specifically, in terms of precision, the animals and the gunshots datasets are the ones that suffer the most with approximately a 5.5% loss. For the recall, the gunshots dataset is degraded by 5% and the birds dataset by 3%. In terms of the F1-score, we see without surprise that the gunshots and animals datasets are the ones having the biggest loss (respectively 5.8% and 3.5%), followed by the birds dataset with 2.4% and the chainsaws dataset with 0.8%.

For the remainder of this Chapter, the *CMSIS-NN* network with the best activations quantization scheme is chosen.

7.4.2 Runtime execution of the *CMSIS-NN* network

For this experiment, we use the microcontroller and the CoreSight debug port that contains a 32-bit counter that counts CPU clock cycles. The explanation and setup to use it to measure the execution time of each layer are found here [206]. We perform the inference measurement on 100 different test samples.

The mean inference time of the convolutional neural network embedded in the microcontroller is shown in Table 7.7. As it can be seen, the total inference takes a bit more than 1% of the initial inference time we have at our disposal (1 second). This is thus a nice result since it would allow the microcontroller to go in deep sleep mode 99% of the time to save power. However, care must be taken since the real-world device should also handle the microphone data and it should include the feature processing part, as discussed in Chapter 8. Analyzing the time it takes for each layer, we see that almost 50% of the inference time is occupied by the first depthwise spatial convolution. The reason is due to the nonuse of the SIMD feature for this first layer because the number of input channels is not a multiple of 4. Other than that, the fully connected layer plays a tiny role in the runtime execution (0.57%) while the convolution layers take almost the entire inference time (97.99%), as it is often the case [207, 208]. It is even more true in our network since we have only convolution layers and a tiny fully connected layer.

Layer	Number of cycles [/]	Time [ms]	Proportion of total runtime [%]
Depthwise Convolution 1	261157	5.441	47.81
Pointwise Convolution 1	50999	1.062	9.34
Relu activation 1	5278	0.110	0.97
Depthwise Convolution 2	117382	2.446	21.49
Pointwise Convolution 2	32524	0.678	5.96
Relu activation 2	1316	0.027	0.24
Depthwise Convolution 3	29320	0.611	5.37
Pointwise Convolution 3	14634	0.305	2.68
Relu activation 3	656	0.014	0.12
Depthwise Convolution 4	12632	0.263	2.31
Pointwise Convolution 4	16525	0.344	3.03
Relu activation 4	462	0.010	0.08
Fully Connected layer	3102	0.065	0.57
Softmax activation 1	166	0.003	0.03
Total	546153	11.378	100

Table 7.7: Runtime execution of the network in *CMSIS-NN*. The results have been obtained as the mean on 100 different inferences. The sum of the timing of each layer is not exactly equal to the total time due to the rounding at 3 decimals values.

The Apollo3 Blue microcontroller contains a 16 kB flash cache that can be enabled for optimization purposes. Three different configurations of this cache are possible as mentioned in [209, page 169 Table 220]. We choose the *direct-mapped configuration with 512 entries*. An explanation of the different cache architectures can be found here [51, page 128 to 131]. The flash cache stores instructions and data that are most used by the processor. It allows to provide single-cycle read access to the ROM and to reduce overall accesses to the latter in order to reduce power. Indeed, flash accesses are slow and costly compared to SRAM accesses [210]. Therefore, enabling the cache should give latency speedups and power consumption reductions. However, it should be noted that the cache is a power-consuming module as well and knowing if it allows reducing the overall power consumption of the device is discovered by experimenting, as it is done in the next subsection.

The analysis of the inference time when this cache is enabled is shown in Table 7.8. The same conclusions can be drawn as for the inference time without the cache. The big change, in this case, is the huge speedup, by a factor of 2.55, of the convolutional neural network inference time due to the cache.

Layer	Number of cycles [/]	Time [ms]	Proportion of total runtime [%]
Depthwise Convolution 1	93313	1.944	43.57
Pointwise Convolution 1	17634	0.367	8.24
Relu activation 1	2356	0.049	1.10
Depthwise Convolution 2	50469	1.051	23.57
Pointwise Convolution 2	14748	0.307	6.89
Relu activation 2	588	0.012	0.27
Depthwise Convolution 3	12640	0.263	5.90
Pointwise Convolution 3	6480	0.135	3.03
Relu activation 3	316	0.007	0.15
Depthwise Convolution 4	5821	0.121	2.72
Pointwise Convolution 4	7939	0.165	3.71
Relu activation 4	268	0.006	0.13
Fully Connected layer	1459	0.030	0.68
Softmax activation 1	135	0.003	0.06
Total	214166	4.462	100

Table 7.8: Runtime execution of the network in *CMSIS-NN* when the flash cache is enabled. The cache characteristics are the following : direct-mapped, 128-bit line size, 512 entries (4 SRAMs active). The results have been obtained as the mean on 100 different inferences. The sum of the timing of each layer is not exactly equal to the total time due to the rounding at 3 decimals values. The sum of the proportions of the total runtime of each layer is not exactly equal to the total proportion due to the rounding at 2 decimals values.

7.4.3 Power consumption of the device

This final experiment allows characterizing the power consumption of the Apollo3 Blue microcontroller when using the designed convolutional neural network. Concretely, the application running on the microcontroller is the following :

1. An input feature is read from the ROM and put into a buffer ;
2. The inference is performed ;
3. The microcontroller goes into deep sleep mode after the inference is done ;
4. The microcontroller is wakened up every 1 second by an interrupt of the Real-time clock [\[211\]](#) ;
5. We go to step 2.

The application is represented graphically in Figure [7.17](#).

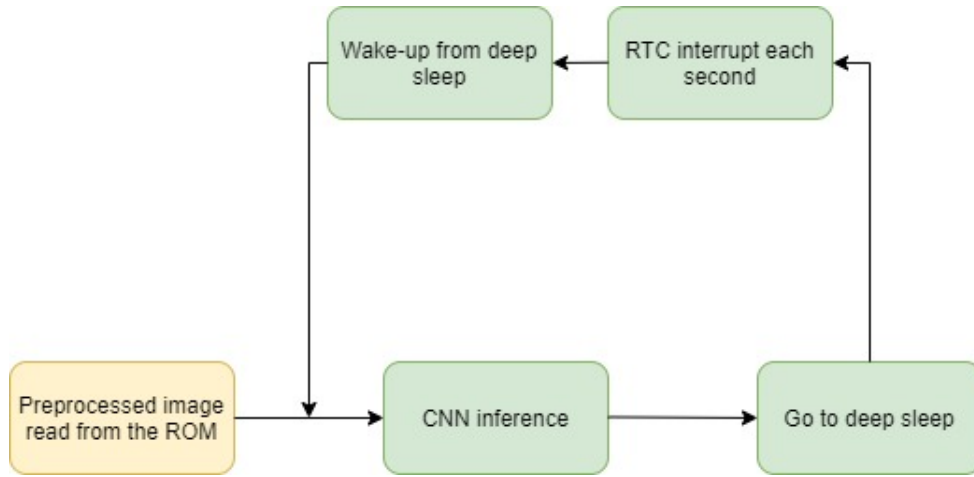


Figure 7.17: Application running on the microcontroller to measure the power consumption of the latter.

To perform this experiment, we need a consistent setup. We use a laptop connected by a USB port to the microcontroller to program it. We also use the *Keithley 2450* Source Measurement Unit [212] to measure the current flowing through the microcontroller. We finally have the *GDS-1052-U* oscilloscope from GwINSTEK [213] that allows, in this setup, to probe one GPIO from the microcontroller. The probed GPIO is in the application reset to 0 when the microcontroller goes to deep sleep mode and set to 1 when the interrupt wakes up the microcontroller to make the inference. Therefore, it allows seeing the correct behavior of the application moving between active mode and deep sleep mode. The setup is shown in the following figure.

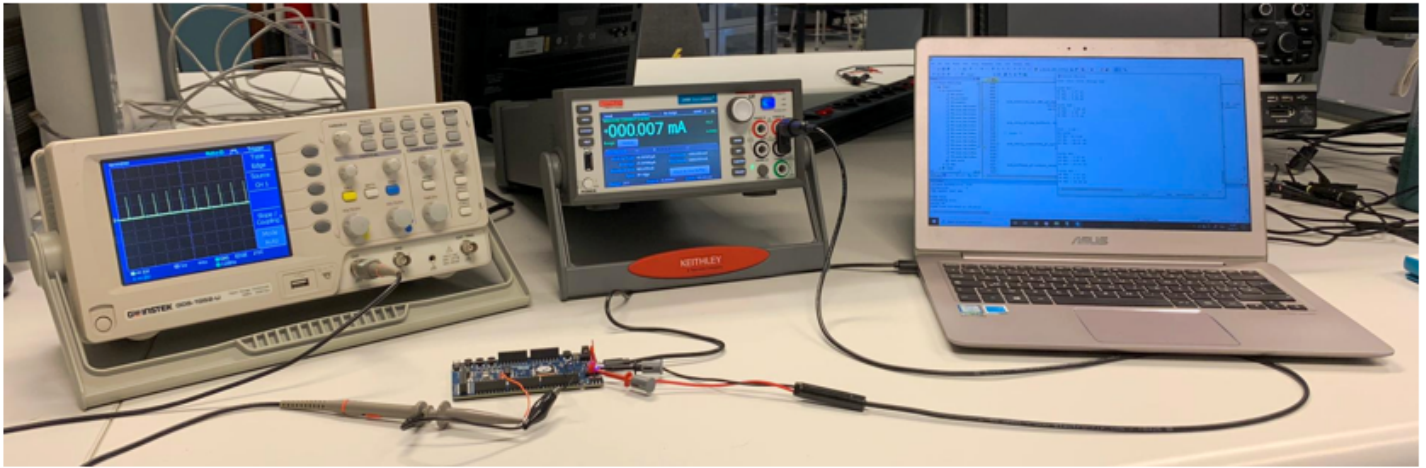


Figure 7.18: Setup used to measure the power consumption of the Apollo3 Blue microcontroller. The output of the oscilloscope shows the active-deep sleep mode variations with a large part of the runtime in deep sleep mode (GPIO set to 0) due to the short inference time compared to the second we have at our disposal.

To reduce as much as possible the power consumption of the device, we play on several features of the microcontroller. Firstly, all the unused peripherals are disabled. Secondly, because the convolutional neural network only uses 3kB of RAM, the smallest SRAM memory of the microcontroller (32kB) is kept while the others are disabled in active and deep sleep modes. It allows removing the leakage power of the unused memories. Finally, we enable clock gating on the 32kB SRAM. Basically, the clock gating technique allows to remove the clock signal when the circuit is not in use allowing to

save dynamical power [214].

The microcontroller contains different configurations. In detail, the clock frequency of the processor can be set to 24, 48, or 96 MHz. The cache can be disabled or enabled according to three possible configurations. Finally, the power supply of the microcontroller can be set to 3.3V or 1.8V. By testing every configuration of the microcontroller, we obtain the power consumption of the microcontroller in Figure 7.19.

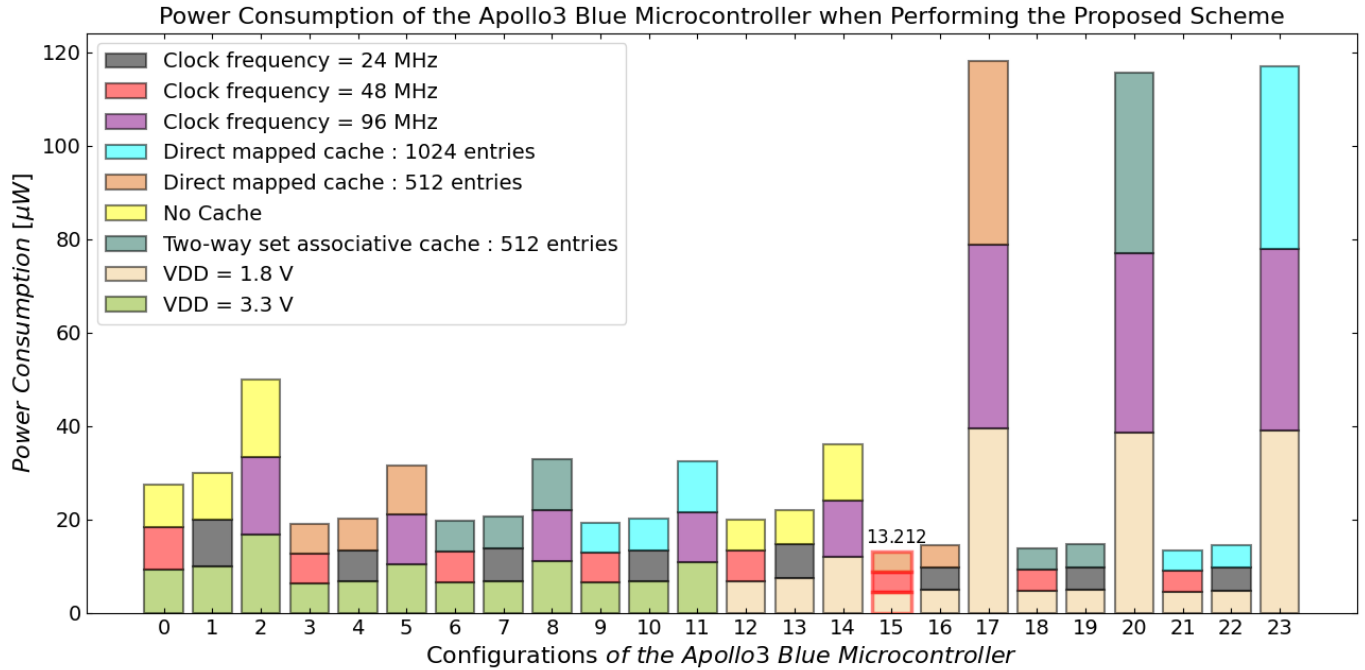


Figure 7.19: Power measurements results depending on the configuration of the microcontroller. Each bar has three colors that are the three parameters used in the configuration.

As it can be seen, the smallest power consumption of the device is 13.212 μW with a clock frequency of 48 MHz, the direct-mapped cache with 512 entries, and a power supply set to 1.8V. To analyze the reasons, we need to know that the power consumption comes from three main sources : the switching power, the short-circuit power, and the leakage power. The first and second sources are dynamic, whereas the last one is static, meaning that it happens even when the circuit is off while the first ones happen only when the circuit is on. Concretely, the switching power occurs when the clock toggles and precisely when a capacitive node is charged. Its equation is given here :

$$P_{SW} = C_L \times V_{DD}^2 \times f_{clk} \times \alpha_F \times N_{nodes} \quad (7.9)$$

where C_L is the load capacitance in Farads, V_{DD} is the power supply of the circuit in Volts, f_{clk} is the clock frequency of the processor in Hz, α_F is the activity factor which is a number between 0 and 1 representing the probability of output switching of the circuit and N_{nodes} is the total number of nodes in the circuit.

The short-circuit power, also called transient power, comes from the current that flows only when the transistors of the devices are switching from one logic state to another. It is basically the power dissipated by an instantaneous short-circuit connection between the supply voltage and the ground when the gates switch their state. It is usually a small fraction of the switching power (5-10% [215])

and is thus neglected in this computation.

Finally, as explained above, the leakage power (static source of power consumption) happens inherently even if the circuit is off. The dissipated power is much smaller than the dynamical power. Its equation is not given because it requires deep knowledge of the fundamentals of transistors. The equation can still be found here [216].

Let us now analyze the results of the power consumption experiment :

- **48 MHz** is the optimal clock frequency. Indeed, at 24 MHz, all factors from Equation 7.9 are the same except for the clock frequency that has been decreased. Therefore, the dynamical power following Equation 7.9 should be divided by two. However, because the runtime execution of the application at 24MHz is twice longer, the dynamical power stays in practice the same. Nevertheless, because the inference time is longer, the leakage power is also bigger (the leakage power in deep sleep mode is extremely small, around $1\mu A$ [209]), thus increasing the power consumption by a small value. At 96 MHz, the microcontroller enters a TurboSpot mode. In that mode, the power supply is internally increased, explaining the big difference in power consumption because the dynamical power explodes ;
- **1.8V** is the optimal power supply. Indeed, due to Equation 7.9, we see that the power supply factor is the most significant one. Therefore, decreasing it should reduce a lot the power consumption ;
- The **direct-mapped cache with 512 entries** is the most optimal. Compared to the other configurations, it uses less active SRAMs, decreasing therefore the power consumption [209]. We also see that it is better to use a cache. The simple reason is that the 2.55 runtime gain allowing to go faster into deep sleep mode is more efficient than the power consumption of the cache itself.

7.5 Summary

In this chapter, we first introduced the *ARM Keil MDK* software development tool that has been used to create the application running on the Apollo3 Blue microcontroller. Then, an in-depth explanation of the ARM *CMSIS-NN* library has been proposed. Due to the use of SIMD instructions, this optimized C library for neural network inference on ARM Cortex-M processors allows a **4.6** $\times$ improvement in runtime and a **4.9** $\times$ improvement in energy efficiency over the baseline ARM *CMSIS-DSP* functions. We also analyzed the *im2col* algorithm used in the convolution *CMSIS-NN* functions that speed up the convolution operation by a factor of more than **200** by transforming the convolution into a matrix multiplication. Moreover, we learned that the *CMSIS-NN* kernel is divided into two branches, the **NNfunctions** allowing to implement popular neural network layer types and the **NNSupportFunctions** being utility functions. We finally saw that the *CMSIS-NN* library works only with 8-bit or 16-bit data types. To that purpose, we explained the fixed-point arithmetic used by *CMSIS-NN* by introducing the Q format of a number.

The next part of this chapter aimed to explain the complete process to convert a machine learning model into its *CMSIS-NN* implementation. The procedure is summarized here :

1. We analyzed the compatibility between the network and the *CMSIS-NN* functions and we explained the need to translate the network into existing *CMSIS-NN* functions ;

2. We transformed each depthwise separable convolution layer, if present, into a depthwise spatial convolution layer followed by a pointwise convolution layer. It allows to ease the process later by having access to the distribution of activations at the output of these layers ;
3. We fused the batch normalization weights into the previous convolution layer if the batch normalization layer was before the activation layer in the network. Otherwise, we fused the batch normalization weights into the next convolution layer. The reason for this folding is due to the lack of a *CMSIS-NN* batch normalization function ;
4. To fit the special weights ordering of the *CMSIS-NN* functions, we had to reorder the weights of specific layers. We reordered the convolution weights into the **NHWC** format and the fully connected weights into the **NC** format ;
5. Because *CMSIS-NN* only works with fixed-point data types, we quantized the weights of each trainable layer with 8 bits following an algorithm minimizing the first-order approximation error (the difference between the quantized weights and the initial ones in floating-point) ;
6. For the same reason as step 5, we quantized the activations of each trainable layer with 8 bits. For that purpose, we computed the activations histograms of each layer and we chose different quantization schemes allowing us to obtain the output Q format of the activations for each layer. The choice of multiples quantization schemes is not mandatory but allows to compare the results when the final implementation is generated ;
7. Because all values are represented as 8-bit or 16-bit integers in *CMSIS-NN*, a system of shifts is essential to adjust the difference between quantized data that do not have the same Q format. We therefore computed the bias and output shifts of the convolution and fully connected layers ;
8. We generated the *CMSIS-NN* implementation in C language using four different files (**main.cpp** ; **parameter.h** ; **input.h** and **weights.h**) for clarity. The last file contains the quantized weights as well as the bias and output shifts of each layer. The third file is optional and contains the input feature used for the inference of the convolutional neural network. The second file contains the specific parameters depending on the architecture of the network. The first file is the main one containing the neural network implementation with the *CMSIS-NN* functions. We also saw how RAM could be saved by switching between two buffers during the inference of the network.

This entire procedure applied on our network allowed to generate a functional *CMSIS-NN* implementation requiring **3040** bytes of RAM and **2014** bytes of ROM.

Finally, this chapter presented the metrics obtained with the *CMSIS-NN* implementation of the optimized model. It reaches **91.37%** of accuracy on the test set, limiting the loss of accuracy to **2.1%** compared to the floating-point network. The embedded convolutional neural network infers a sample in **4.46** ms using the flash cache. The power consumption of the Apollo3 Blue microcontroller is **13.212 μW** when looping between CNN inference in active mode and deep sleep mode.

This work comes with a limitation due to the larger context it is part of. Therefore, this chapter aims to introduce this limitation and discuss interesting issues, one of them due to the larger context of the work. Finally, we compare our work with the state-of-the-art.

8.1 Limitation and larger context

In this Master thesis, we cannot yet deploy, in the forest field, the convolutional neural network that has been embedded onto the Apollo3 Blue microcontroller. Indeed, we implemented the feature processing block in software on the setup machine, and we streamed the features afterward on the microcontroller. However, it should not be the case in a real-world implementation as the feature processing block would be implemented on the board or another chip. We show a proposal of a block diagram for the complete circuit required to classify forest sounds among the animals, birds, chainsaws, and gunshots categories at an ultra-low-power consumption in Figure 8.1. Let us explain it in detail.

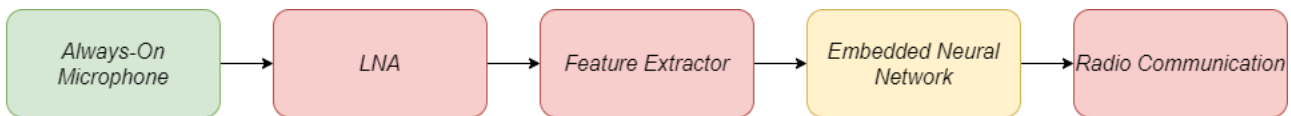


Figure 8.1: Overview of the design outside this Master thesis using our embedded software convolutional neural network. The green block is the always-on microphone designed by [217], the yellow block is our embedded convolutional neural network, and the red blocks are the remaining parts.

The first element of this block diagram is an *Always-On Microphone*. An active microphone constantly listens to the forest environment in a frequency range between [90; 16000]Hz, translating the analog waveform into an electric signal. Based on an energy detection of the sound, the analog front-end part of the block captures only the relevant sounds that are further analyzed in the remainder of the implementation. We reject the other sounds if they do not pass the given calibrated energy threshold. The microphone consumes a current of $16\ \mu A$, while the analog front-end allowing the detection of relevant sounds consumes exactly $1\ \mu W$ of power. The design is based on the work of [217], done simultaneously as this thesis.

The second element is an efficient *Low-Noise Amplifier* [218], allowing to amplify the input signal for the remaining blocks. The performance of the LNA would be of major interest in order not to degrade the original signal-to-noise ratio of the sound. Furthermore, studies would be dedicated amongst others to optimize the minimum detectable signal and the total harmonic distortion. The latter is crucial since a higher percentage would yield more harmonic distortion in the signal. This is not a property we want in our system since it would degrade the spectral content of the sound, which is of great interest for the classification.

The third block would be the feature extractor allowing to obtain the MFCC feature from the original sound. For the moment, we implemented the feature extraction in software on the setup machine, but the goal is to make it in the analog domain as a hardware accelerator or in the digital domain as a digital signal processor. It would increase its energy efficiency.

The fourth block is the embedded neural network designed in this work, allowing to make the classification of sounds. It would take the MFCC feature directly obtained from the feature extractor block and use it for the inference.

Finally, the last block is radio communication. After the inference, the sensor must send the outputted class to the cloud. Otherwise, the information could be lost. From that point, the cloud could send alerts to forest guards if a gunshot or a chainsaw sound has been detected in the area. The radio communication block has partly been studied by another Master thesis [219], focusing on *LoRa* communication. Its features are a long-range ultra-low-power communication with a limited bit rate.

Several examples of state-of-the-art works following this block diagram and based on application-specific integrated circuits are presented hereafter. It proves that the illustrated block diagram in Figure 8.1 follows state-of-the-art implementations: an always-on keyword spotting chip consuming 510 nW is proposed in [220]. It uses a serial-FFT-based Mel-frequency cepstrum coefficient analog circuit to extract the features to be inferred through a shallow binarized depthwise separable CNN implemented as a neural network accelerator. The authors use a 40kHz clock frequency allowing a power supply of 0.41 V. The classification accuracy on a one-word KWS task reaches 97.3% with the Google speech command dataset. In the same trend, [221] demonstrates a background-noise and process-variation-tolerant 109nW acoustic feature extractor chip based on spike-domain divisive-energy normalization. They pair it with a spiking neural network classifier chip consuming 570 nW to create a keyword spotting system achieving 89 to 94% of accuracy. Another work presents a 12.2 nW always-on acoustic sensing and object recognition microsystem [222]. It basically embeds the feature extraction and a Support-Vector Machine classifier as dedicated digital signal processors. The microsystem reaches more than 95% of detection on a dataset of generator, car, and truck sounds. Finally, an energy-efficient architecture to extract MFCC features in the analog domain is presented in [223]. It allows extracting the features directly in the analog domain by consuming $0.72 \mu J/frame$ and can be used in any implementation.

8.2 Interesting issues

8.2.1 Sample rate review

In Section 4.4.2, we obtained an optimal sample rate of 16 kHz for the targeted classification task. The optimization criterion was based on the limitation of information loss on the frequency content of the birds and animals dataset. However, because of the ultra-low-power feature we seek, it could

be interesting to review this sample rate value and analyze how the number of operations of the final model scales with it. This is shown in Figure 8.2. We observe almost a linear relationship between the sample rate and the number of MACC operations of the model. Concretely, as explained in Section 2.6.3.1, the sample rate only affects the number of temporal frames obtained after the STFT process following Equation 2.14. Therefore, it modifies the input feature size, which plays an important role in the number of MACC operations of the model. Relaxing this sample rate constraint can be helpful to reduce the power consumption of the embedded CNN, and the one of the microphone by decreasing its frequency bandwidth. However, it would induce a loss of classification accuracy due to the loss of frequency information. To quantify precisely the impact, a retraining of the model must be done with the new feature size due to the different sampling rate. If the accuracy loss is acceptable, it could allow an important reduction of the power consumption of the entire chain.

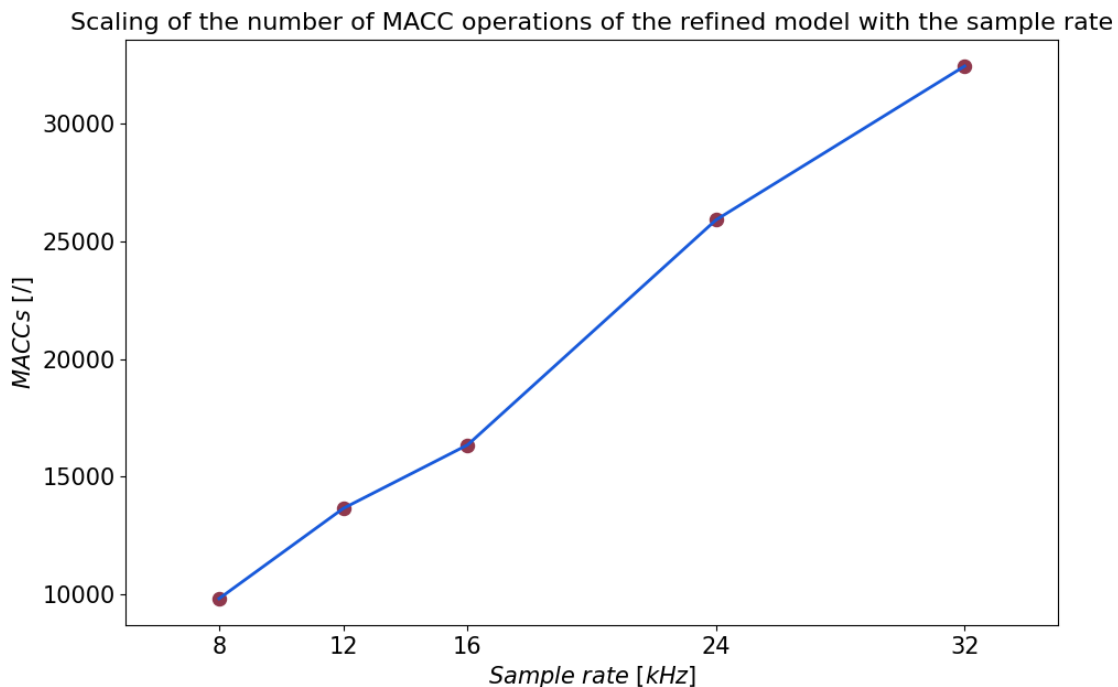


Figure 8.2: The scaling of the number of MACC operations of the optimized refined model with respect to the sample rate of the recordings demonstrates a behavior close to the linear one.

8.2.2 Segmentation technique review

As highlighted in Section 4.3, the segmentation technique used in this work during the pre-processing step of the *Events* dataset is not optimal. It requires the assumption of a quiet environment for which the sound of interest is higher in intensity than the background noise. However, this is not always true in practice. Simple examples of forest events are big rainfalls or powerful winds. In that specific case, the presented segmentation technique considers background parts of the recording as relevant. We want to avoid this process since it could lead to unwanted 1-second slices used in the training process. Therefore, we discuss another possibility to improve the segmentation process.

The improvement is based on [224]. It allows making a multi-class environmental sound segmentation using a spectrogram as input. The principle is represented in Figure 8.3, for which 75 categories can be classified. We see that the three classes defined in the spectrogram of the recording are correctly classified at the right moment. The work uses what they call the *Mask U-Net* deep learning architecture, which combines segmentation using the U-Net network and sound event detection using

a convolutional neural network. The *U-Net* topology is a fully convolutional network that has proven to be very effective in biomedical image segmentation tasks [27]. The basic principle of U-Net is to obtain a lower-dimensional representation of the image and then upsample that low-dimensional representation to produce the final output segmentation map. The *Mask U-Net* network, fully described in [224], predicts a mask allowing to separate each class in the spectrogram of the original sound. Inverse Short-Time Fourier Transform can then be applied to this mask to reconstruct the different audio signals from each category. Note that because the phase is not present in the mask, we must take it from the original sound for the reconstruction. In fact, this work can be applied as our segmentation technique allowing, for instance, to separate two classes: important sound and noise. The important sound class would contain bird, gunshot, and animal sounds, whereas the noise part would contain any possible forest noise, including heavy rainfall or powerful wind. The network should, of course, be trained for that task, and a non-negligible time would be dedicated to the correct labeling of training samples. Nevertheless, the performance of this segmentation technique could be remarkable, and the authors are still working on it to improve their baseline solution [225].

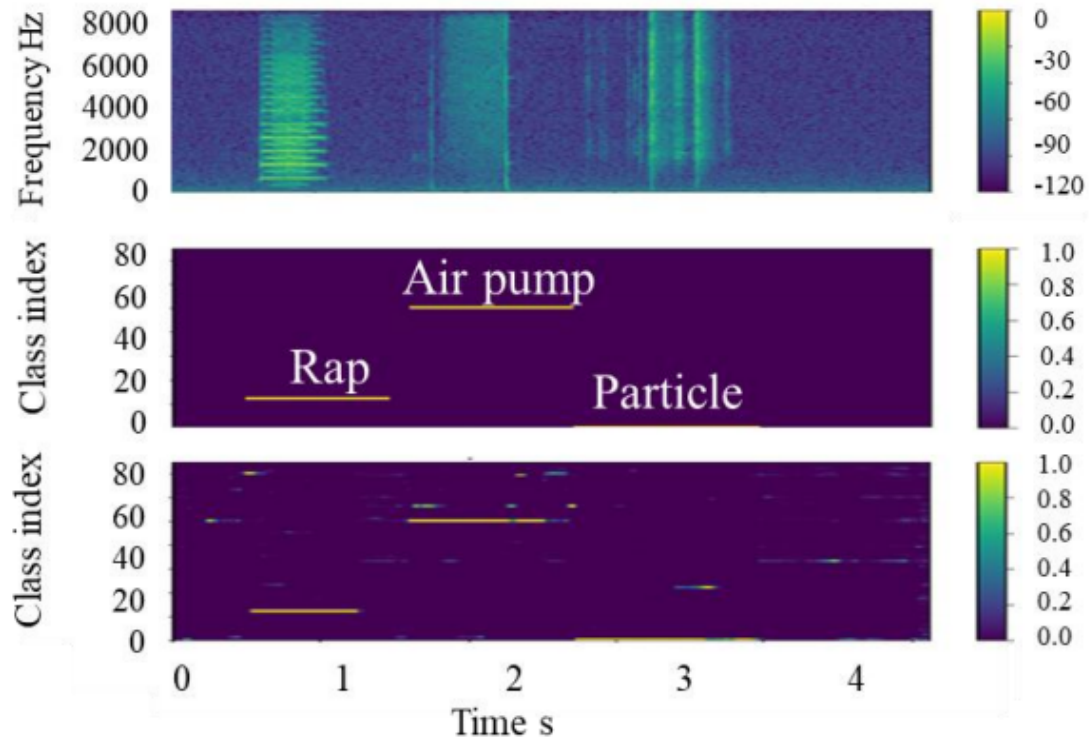


Figure 8.3: An example of sound event detection. The upper panel shows the input spectrogram, the center panel shows the ground truth, and the lower panel shows the prediction [224].

8.2.3 Model analysis

For many practitioners, convolutional neural networks are considered a black box for which it is not trivial to explain the decisions of the network that are made to classify a given sample. For a few years, several works have tried to demystify the internal behavior of the convolutional neural networks. [226] allows visualizing all the pixels that drive the choice made by the network. They ask themselves if the model truly identifies the important object or part in the image or uses the surrounding context. To that purpose, they occlude with a grey square different portions of the input image, and they analyze each time the output of the classification process. The result for three different samples from the *ImageNet* dataset is shown in Figure 8.4. The first row is explained in detail:

- First column : the input image represents a Pomeranian, a breed of dog. We occlude with a grey square a specific portion of the image and we do that for the entire image ;
- Second column : the probability of correct class of the classifier is represented as a function of the position of the grey square. We see that all the outline of the image (in the second column) is red, meaning that occluding the surrounding context does not harm the classification output. However, obscuring the dog's face (middle area in blue) makes the correct probability drop a lot. The CNN thus takes a wrong decision. It proves that the CNN can use the important part of the image in its decision, being the dog's face in that example ;
- The most probable class is given as a function of the position of the occluder. In most cases, the most probable class is the Pomeranian (dark blue on the image). However, for instance, hiding the dog's face but not the tennis ball involves a bad decision of the classifier. The classifier's most probable class becomes the tennis ball (light blue). It proves again that the classifier uses the important information in the image to make its decision.

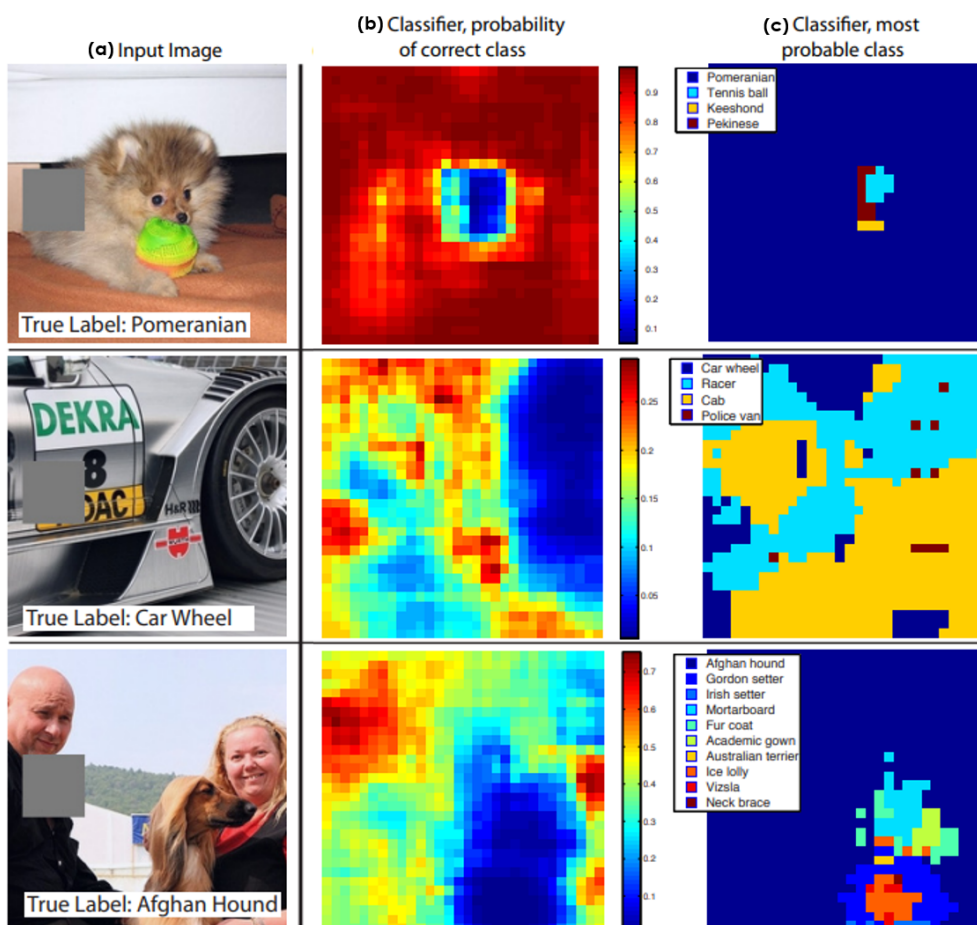


Figure 8.4: Three test samples from *ImageNet* systematically occluded with a grey square in the first column. The second column represents the probability of the CNN to output the correct class as a function of the occlusion position in the input image. The red color represents a high probability, whereas the blue color represents a low probability of correct class. The third column shows the most probable class to be outputted by the CNN with respect to the position of the grey square [226].

Another recent work proposes a technique called Gradient-weighted Class Activation Mapping (Grad-CAM) that produces a localization map highlighting the important regions in the image for predicting the correct class [227]. An example is shown in the second column of Figure 8.5 where the important pixels of the cat and dog classes are highlighted in different maps allowing a deep visualization of the decision of the neural network.

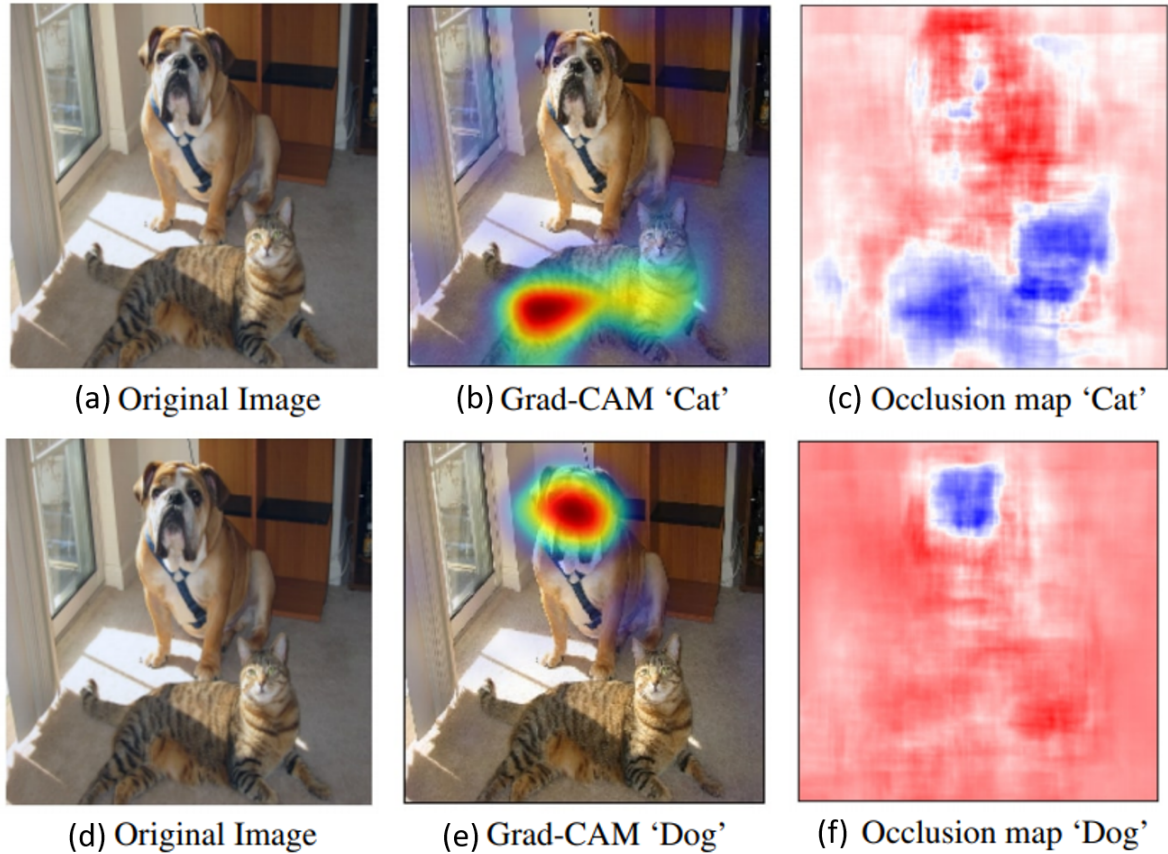


Figure 8.5: One test sample represented in the first column. The second column represents a map showing the discriminative parts of the image for a specific class. The last column proves that the GRAD-CAM technique provides similar results as occlusion sensitivity while being much cheaper to compute [227].

Based on these two works, it could be interesting to implement one of those on the refined model to see if the convolutional neural network uses the correct information from the MFCC feature to drive its decision.

8.2.4 Microphone detection reliability

As explained in Section 8.1, the microphone has been designed in hardware as an always-on device to capture only relevant sounds. It is supposed to extract from the forest environment only animal, chainsaw, bird, or gunshot sounds since they are the categories to be classified in this large work. However, the detection compares the energy of the sound with an energy threshold calibrated for forest sounds. It is unlikely that this energy-based detection works one hundred percent of the time. For instance, thunderstorms or wind can lead to high energy sound exceeding the calibrated threshold. In this case, it would mean that an irrelevant sound would be considered as relevant and therefore preprocessed and inferred through the convolutional neural network. Because the sound

is an out-of-distribution sample with respect to the four categories to be outputted by the neural network, it means that we would consume energy for an irrelevant task. To cancel or decrease this problem, we propose two solutions with their specific trades-off:

1. Train a new convolutional neural network with one more output class named **noise**. This class would be the one in which we would put irrelevant samples to be classified. The difficulty is that it is not easy to classify all types of irrelevant sounds into a single class. However, in this work, we obtained many environmental noise data from *Rainforest Connection*, which contain multitudes of different sounds, making it possible to create this "catch-all" class. However, the trade-off is that it would result in a bigger neural network because of the new output neuron in the last layer, leading to a higher consumed energy. Moreover, if this solution is chosen, we could not use the environmental data from *Rainforest Connection* for the data augmentation since it would be used as training samples for the new class ;
2. Detect out-of-distribution samples. As explained in this paper [228], it is possible to detect misclassified and out-of-distribution samples in a neural network using the softmax probability of the predicted class. The advantage is the possibility to keep the smaller 4-classes CNN used in this work. However, the out-of-distribution sample is found only after the inference meaning that we would consume power for an irrelevant sound. Furthermore, this technique has been the first study in this domain and is considered as a baseline. Indeed, the results it yields are far from perfect, and the authors stated that we could make a lot of improvements.

Several studies have been made following [228] to improve the baseline, which has been found. [229] makes the detection more reliable and efficient. Their method is based on the observation that using temperature scaling and adding small perturbations to the input can separate the softmax score distributions between in- and out-of-distribution images, allowing for more effective detection. It uses two inference passes: the first one is used to find the out-of-distribution sample, and the second one is used, in case the sample is not an out-of-distribution sample, to find the output class. The two networks are the same, but two passes are necessary since the input image is not preprocessed in the same way to see if it is an out-of-distribution sample and to infer. The advantage of this technique is that it improves the baseline by a large margin: for instance, their method reduces the false positive rate from 34.7% for the baseline to 4.3% on the DenseNet (applied to CIFAR-10 and Tiny-ImageNet) when the true positive rate is 95% [229]. However, it doubles the power consumption of the inference and the latency of the task. Another work tackling the out-of-distribution detection problem and claiming to surpass the above-proposed techniques can be found here [230].

8.3 Comparison with the state-of-the-art

This section allows comparing our work with state-of-the-art implementations related to audio monitoring of forest ecosystems. However, because the targeted task has never been done in the literature and does not contain benchmark datasets, it is complicated to make the comparison. Therefore, we have decided to split our comparison into two parts : related audio forest monitoring works and ultra-low-power CNN works. The first part allows knowing the state-of-the-art accuracies in chainsaw and gunshot audio detection, while the second part allows emphasizing on the power performance obtained by our work.

Table 8.1 shows the results obtained by several works in chainsaw and gunshot audio detection.

- Chainsaw detection : we see that our system yields the second-best performance. Several strong aspects can be discussed as part of these results. First, our final network allows an ultra-low-power consumption and is quantized and embedded on a microcontroller, yielding a chainsaw recall loss of 1.6% compared to the floating-point version and allowing real-time sound classification on the field. Secondly, the test set we use to obtain the recall on the chainsaw class is composed of 26 570 audio recordings. However, it is not the case for the presented state-of-the-art works : 38 testing chainsaw samples are used by [7], 487 samples are used both for training and testing the network for [231], and 50 chainsaw testing samples from [232]. Therefore, the robustness of the classification can be discussed, whereas our work uses a large testing dataset coming from different forest areas, proving its efficiency.
- Gunshot detection : Our work yields the lowest recall for the gunshot class. However, as before, we compare it with floating-point complex networks explicitly trained for a binary classification task, which is not our case. Using the floating-point ultra-low-power version achieves a recall of 96.3%, taking second place in the ranking.

Work	Task	Test positive recall [%]
S. Segarceanu (2020) [7]	Chainsaw detection	72.77
G. Nicolae (2018) [231]	Chainsaw detection	97
J. Daniel C. Arevalo (2020) [232]	Chainsaw detection	100
Our work (2021)	Chainsaw detection	97.9
V. Singh (2020) [233]	Gunshot detection	96.04
S. ur Rahman (2021) [234]	Gunshot detection	95.08
J. Bajzik (2020) [235]	Gunshot detection	98.79
Our work (2021)	Gunshot detection	91.1

Table 8.1: Part 1 : state-of-the-art comparison for chainsaw and gunshot audio detection. The positive test recall corresponds to the recall for the chainsaw or gunshot class.

Because our network has not been trained specifically for chainsaw or gunshot detection, and since it has been built for high energy efficiency, the results we get on these two tasks are satisfactory. We succeed in beating state-of-the-art complex networks for these detection tasks.

Table 8.2 compares our work with different classification tasks, not all audio-based. It aims to compare the embedded CNN performance on a microcontroller between several works. As it can be seen in this table, the designed CNN in this thesis requires a tiny amount of memory below 5 kB. It is one of the factors leading to a power consumption in the $10\mu W$ range. Our work gives the smallest power consumption for neural networks embedded on a microcontroller while reaching a high test set accuracy.

Work	Task	Test accuracy [%]	CNN inference time [ms]	RAM usage [kB]	ROM usage [kB]	Device power [μ W]	Core frequency [MHz]
J. Norby (2019) [30]	Environmental sound classification	70.9	43	54	180	-	80
F. Naccari & al. (2020) [236]	Acoustic scene classification	89.17	36.022	4.94	7.71	-	80
M. Meyer & al. (2019) [237]	Seismic event classification	97.6	440	85.6	38.403	2958	80
M. Eggimann & al. (2019) [238]	Gesture recognition	92	-	-	92	21000	100
G. Cerutti & al. (2019) [239]	People presence detection	76.7	2	6	20	2300	80
Our work (2021)	Forest sounds classification	91.37	4.46	3.04	2.014	13.212	48

Table 8.2: State-of-the-art comparison in terms of embedded CNN performance.

We can improve this work in several ways. Some insights are therefore presented in this chapter.

9.1 Bird species classification

Classifying bird species would be very beneficial for wildlife conservation. To that purpose, we could create a second ultra-low-power model on the embedded device. This network would be cascaded with the one made in this work and would be enabled if the inferred class of our convolutional neural network is "birds". We show the block diagram in Figure 9.1.

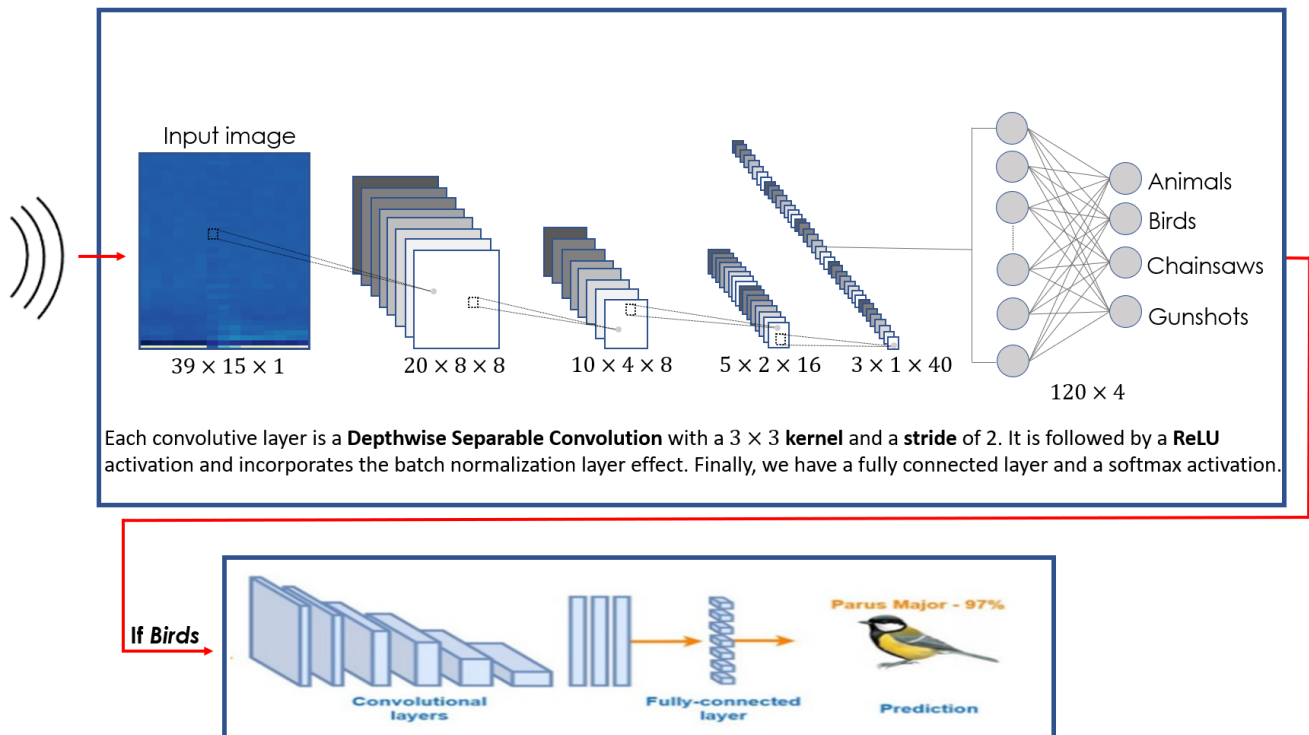


Figure 9.1: The overview of the two convolutional neural networks on top of each other. The first CNN on the left is the one created in this work, while the second on the right is the improvement allowing bird species classification. The picture on the right of the figure comes from [240].

The activation of this second model with a fixed criterion is a clever choice to minimize the power consumption of the entire device. Moreover, because we would not use the networks simultaneously, the random access memory could be reused and overwritten for the inference of the second model. Because the bird species classification task is more difficult [140], this new network could be more complex, giving rise to a higher power consumption. However, since we would enable the model only a small fraction of the time (when a bird sound is detected), the added power consumption would be small compared to the first network. This improvement is possible because the Apollo3 Blue microcontroller used in this work contains enough Flash memory to embed a new network. We can find a good starting point to bring this idea to life in [241].

9.2 Wildfires detection

Wildfires are another important and interesting phenomenon that destroy forest ecosystems [242, 243, 244]. Therefore, detecting them in an early stage would be highly relevant since it would allow an immediate intervention to limit the enormous damages they create. However, an audio dataset on wildfires is almost impossible to find in large quantity, making the audio classification task unsolvable. To fix this issue, we could use sensors to capture important information related to wildfires. Such sensors could be based, for instance, on temperature, smoke, or gas concentration. Because the embedded device must be ultra-low-power, it would be better to make the complete design of the sensor(s) from scratch to limit as much as possible their power consumption. If it is not possible, we can use the embedded sensor(s) of the microcontroller with an additional cost in power that has to be taken into account. Based on the data captured by the sensor(s), an ultra-low-power machine learning model could be built and deployed specifically to detect wildfires. A lot of works related to the detection or prediction of wildfires exist in the literature. An inspiring solution, using sensors data and images coupled with machine learning, can be found in [245].

9.3 Multi-label classification

In this work, we perform a single-label classification task because we can infer only one class. However, multi-label classification allows inferring multiple classes on a 1-second audio slice. It would be closer to reality since a chainsaw sound can happen simultaneously than a bird song, for example. The same idea applies to bird species classification, for which multiple birds could sing at the same moment.

9.4 Refinement of the embedded device and library

As highlighted in Section 7.2, *PULP-NN* is another library that could be used in this work, replacing *CMSIS-NN*. With the GAP-8 processor running at the maximum clock frequency, *PULP-NN* claims to outperform the *CMSIS-NN* Cortex-M4 reference implementation by a factor of 14.1 in terms of energy efficiency [191]. Using this library and the GAP-8 processor could thus allow decreasing the power consumption of the device.

Forest ecosystems are a living space for millions of animals and highly contribute to human species preservation. However, many illegal activities have become more present in forest environments : large-scale deforestation destroys animal habitats and contributes to global warming, while animal poaching threatens many species with extinction. Therefore, it becomes paramount to continuously and automatically monitor the forest environments to prevent illegal activities from happening. Forest ecosystems audio monitoring is also helpful to monitor animal species living in a specific area of the globe. To fulfill these objectives, the focus of this Master thesis was to build an embedded ultra-low-power convolutional neural network on the Apollo3 Blue microcontroller, classifying 1-second sounds among the animals, birds, chainsaws, and gunshots categories.

The first primordial step was to gather an audio dataset related to the targeted classification task. The *Xeno-Canto* online database has allowed retrieving **4822** bird recordings, while the *Gunshot Audio Forensics* dataset has provided **1990** gunshot samples. However, we did not find open-source datasets with large animal and chainsaw sound samples. Therefore, two enterprises in the United States, known as *Rainforest Connection* and the *Borror Laboratory of Bioacoustics*, have been requested. The first company provided us with **88 567** chainsaw **2**-seconds recordings and **4909** gunshot audios. The other enterprise gave us its mammal sound database composed of **1099** recordings. The combination of these different databases has created our original audio-based dataset. However, due to the various acquisition sources, the recordings were very different with respect to their audio properties : sample rate, resolution, duration, number of channels, quality, and proportion of noise. Therefore, processing the dataset was essential. To that purpose, 3 main steps have been performed on the recordings :

1. Resampling to **16** kHz. This value was optimal on our dataset based on a loss minimization regarding the frequency content of the animal and bird recordings ;
2. Rewriting each recording with a **16**-bit resolution ;
3. Segmenting each sample into **1**-second slices. Three 1-second cuts have been performed on every 2-seconds chainsaw recording, while a more sophisticated segmentation technique, based on a spectrogram analysis and coming from Bird Classification research works, has been used for the other classes. This technique outputted a mask segmenting the important sounds from the noise parts. Afterward, we traveled over the entire recording and analyzed each 1-second audio slice. If it contained at least **X**% of important sound, with **X** a threshold value, the slice was taken. **X** was chosen to be **15**% for the animals and birds classes, and **35**% for the gunshot samples.

This entire process has yielded a modified imbalanced dataset, referred to as the **Events** dataset, containing **668 746** 1-second recordings at a sample rate of **16** kHz and a **16**-bit resolution.

Using the *Events* dataset, we were able to train a **MobileNetV1** CNN architecture as a baseline model. The network was chosen due to the use of depthwise separable convolutions, decreasing the number of MACC operations significantly compared to a regular convolution. To maximize the accuracy of the model and its energy efficiency, a four-step approach was performed using a **5%** subset of the *Events* dataset :

1. Obtaining the most optimized feature type. The **MFCC** feature computed with a **Hanning** window of **512** samples, **75%** overlap, **128** mel filters, and **39** coefficients was chosen, leading to a validation accuracy higher than **96%** while requiring **13 026 380** model MACC operations ;
2. Performing **Time Shift**, **Noise Injection**, **Pitch Shift**, and **Time Stretch** data augmentation techniques allowed to improve the classification accuracy by **1.19%** on average ;
3. Tuning the training hyperparameters allowed to increase further the classification accuracy by **1.97%** using the **Adam** optimizer, a learning rate equal to **0.001**, and a batch size set to **128** ;
4. The best model was then trained using the full dataset and the results of the previous experiments. It achieved an accuracy of **99.64%** on the test set.

Even though the test set accuracy was close to perfection, the baseline model could not be embedded into the Apollo3 Blue microcontroller. The flash memory of the latter was limited to **1** MB, while the amount of memory required for storing the weights of the baseline model was equal to **3.32** MB. Furthermore, the number of MACC operations of the baseline model exceeded the maximum capacity of the microcontroller to be able to work in real-time. Therefore, a refinement of the architecture has been performed following a five-step approach on the **5%** subset of the *Events* dataset :

1. A study of numerous CNN architectures has been made, yielding a novel architecture based on depthwise separable convolutions. This architectures computed a factor of $\times$ **273** fewer MACC operations than the baseline model and required $\times$ **1655** fewer ROM usage for its weights ;
2. Because the downsampling factor of the refined architecture has been reduced, the MFCC feature set computed with a **Hanning** window of **512** samples, **128** mel filters, and **39** coefficients was chosen. It achieved a factor of 2.77 fewer model MACC operations for a drop of validation accuracy below **1%** ;
3. Performing **Time Shift**, **Noise Injection**, **Pitch Shift**, and **Time Stretch** data augmentation techniques allowed to improve the classification accuracy by **0.74%** on average ;
4. We tuned the training hyperparameters and the number of output feature maps of each convolutive layer. It allowed reducing MACC operations by **5%** while decreasing the classification error by **5.14%**. This result used an **Adam** optimizer, a learning rate of **0.1**, a batch size of **96**, **8** filters in the two first convolutive layers, and respectively **16** and **40** feature maps for the third and last convolutive layers ;
5. Based on the previous experiments, we trained the refined model on the entire *Events* dataset. The optimized refined model achieved **93.5%** of test set accuracy, harming the accuracy by **6.14%** compared to the optimized baseline model. However, it required a factor of $\times$ **797** fewer model MACC operations and a factor of $\times$ **1605** fewer model parameters.

Due to the significant improvements obtained with the optimized refined model, it was possible to embed it on the Apollo3 Blue microcontroller. The *CMSIS-NN* library has been used for that purpose. This is an optimized C library for neural network inference on ARM Cortex-M processors allowing a $4.6\times$ improvement in runtime and a $4.9\times$ improvement in energy efficiency over the baseline ARM *CMSIS-DSP* functions. It required a tedious process to convert the refined model into its *CMSIS-NN* implementation, for which the most important steps are given hereafter :

1. Fusing of the batch normalization weights into the convolution layers ;
2. Convolution and fully connected weight reordering ;
3. Weights and activations **8-bit** quantization ;
4. Computation of the bias and output shifts ;
5. Generation of the *CMSIS-NN* implementation.

Performing this process on our refined model generated a *CMSIS-NN* model, requiring **3040** bytes of RAM and **2014** bytes of ROM. Then, we computed the test set accuracy of the embedded model, its runtime execution, and the power consumption of the device :

- The embedded model achieved a test set accuracy of **91.37%**, limiting the loss of accuracy to **2.1%** compared to its floating-point version ;
- The inference lasted **4.46** ms, using the SIMD feature from *CMSIS-NN* as well as the flash cache of the microcontroller ;
- Experimental measurements of the board have demonstrated a power consumption as little as **13.21 μ W** to do an inference each second. After each inference (99.5% of the time), the deep sleep mode of the microcontroller is used to save power.

Our work has been compared to state-of-the-art implementations. First, we demonstrated state-of-the-art accuracy results on a chainsaw and gunshot audio detection task using our 4-classes embedded CNN. Secondly, we have shown its state-of-the-art energy efficiency compared to several works in the literature involving classification on ultra-low-power microcontrollers. **This work has thus demonstrated the possibility to deploy thousands of ultra-low-power high-accuracy embedded CNNs to monitor forest ecosystems.** Even though the objectives have been filled, it is still possible to improve the work. Enabling bird species classification or wildfires detection is such an example. Moreover, the work can be followed by building a feature extractor block since our embedded CNN currently relies on streamed features for the inference, which is not a real-world case.

Appendices

A.1 Segmentation technique Python code

We first show the function *getSpectrogram* needed to compute the normalized spectrogram with the correct parameters required by the segmentation technique. Then, we present the function *SegmentationTechnique* that uses the normalized spectrogram to create a signal mask out of the audio recording.

```
1 import matplotlib.pyplot as plt
2 import librosa
3 from librosa import display
4 import numpy as np
5
6 def getSpectrogram(audiopath, TestMode):
7     """
8     Compute the normalized spectrogram of the audio file
9     Functions parameters :
10     - 'audiopath' : Audio recording (.wav for instance)
11     - 'TestMode'  : If set to True, it enables the plot of the spectrogram
12                   and the normalized spectrogram
13     """
14     sig, sr = librosa.load(audiopath, sr=None) # preserve the natural sample rate!
15
16     #Parameters
17     N_FFT = 512
18     HOP_SIZE = 384 #75\% overlap
19     WINDOW_TYPE = 'hann' #Hanning window
20     FMIN = 0
21
22     #Process
23     STFT = librosa.stft(sig, n_fft= N_FFT, hop_length=HOP_SIZE,
24                         window=WINDOW_TYPE, center = False)
25     spec = np.abs(STFT)**2
```

```

26
27 #Plot the spectrogram if TestMode enabled
28
29 if TestMode:
30
31     plt.figure(figsize=(13, 3))
32     librosa.display.specshow(spec,cmap='YlGnBu', sr= sr,
33                             fmin=FMIN, hop_length=HOP_SIZE,y_axis='linear', x_axis = 's')
34     plt.title('Original spectrogram', fontsize = 20)
35     plt.ylabel('Frequency [Hz]', fontsize = 18)
36     plt.xlabel('Time [s]', fontsize = 18, position = (0.49,0))
37     plt.tick_params(labelleft=False,labelbottom=False)
38     plt.show()
39
40 #Normalize the spectrogram
41 spec -= spec.min()
42 spec /= spec.max()
43
44
45 #Plot the normalized spectrogram if TestMode enabled
46 if TestMode:
47
48     plt.figure(figsize=(13, 3))
49     librosa.display.specshow(spec, sr= sr, fmin=FMIN,
50                             hop_length=HOP_SIZE, cmap='YlGnBu',y_axis='linear', x_axis = 's')
51     plt.tick_params(labelleft=False,labelbottom=False)
52     plt.title('Normalized spectrogram', fontsize = 20)
53     plt.ylabel('Frequency [Hz]', fontsize = 18)
54     plt.xlabel('Time [s]', fontsize = 18, position = (0.49,0))
55     plt.show()
56
57 #return the normalized spectrogram, the samples of the audio recording,
58 #and the sample rate
59 return spec,sig,sr

```

```

1 import matplotlib.pyplot as plt
2 import librosa
3 from librosa import display
4 import numpy as np
5 import cv2
6 import scipy.ndimage as ndimage
7
8 def SegmentationTechnique(CompleteAudio, CompleteFileSpec, MedianBlur,
9                           MedianBlurPasses, Kernel, Threshold, TestMode, sr):
9     """
10     Return the signal mask of the audio file
11     Functions parameters :
12     - 'CompleteAudio' : Audio data from function 'getSpectrogram'

```

```

14 - 'CompleteFileSpec' : Normalized spectrogram from function 'getSpectrogram'
15 - 'MedianBlur' : If set to True, it enables the use of the median blur filter.
16           It must be true in our case.
17 - 'MedianBlurPasses' : If MedianBlur is True, it gives the number of passes
18           in the filter (5 in our case)
19 - 'Kernel' : Filter used in the opening operation (4x4 in our case)
20 - 'Threshold' : Median threshold (4.5 in our case)
21 - 'TestMode' : If True, it enables multiple plots
22 - 'sr' : Sample rate from function 'getSpectrogram'
23 """"
24
25 ## Working copy
26 img = CompleteFileSpec.copy()
27
28
29 ### STEP 3: Median blur
30 if MedianBlur:
31     img = cv2.medianBlur(img, MedianBlurPasses)
32
33
34 if TestMode:
35     plt.figure(figsize=(13, 3))
36     librosa.display.specshow(img, sr= sr, fmin=0, hop_length=384,
37                             cmap='YlGnBu', y_axis='linear', x_axis = 's')
38     plt.tick_params(labelleft=False, labelbottom=False)
39     plt.title('Median blurring of the normalized spectrogram', fontsize = 20)
40     plt.ylabel('Frequency [Hz]', fontsize = 18)
41     plt.xlabel('Time [s]', fontsize = 18, position = (0.49,0))
42     plt.show()
43
44
45
46 ### STEP 4: Median threshold
47 col_median = np.median(img, axis=0, keepdims=True)
48 row_median = np.median(img, axis=1, keepdims=True)
49
50 img[img < row_median * Threshold] = 0.0
51 img[img < col_median * Threshold] = 0.0
52 img[img > 0.] = 1.0
53
54
55 if TestMode:
56     plt.figure(figsize=(13, 3))
57     librosa.display.specshow(img, sr=sr, fmin=0, cmap='gray_r',
58                             y_axis='linear', x_axis = 's')
59     plt.tick_params( labelleft=False, labelbottom=False)
60     plt.title('Selected pixels', fontsize = 20)
61     plt.ylabel('Frequency [Hz]', fontsize = 18)
62     plt.xlabel('Time [s]', fontsize = 18, position = (0.49,0))
63     plt.show()

```

```

64
65
66
67 ### STEP 5: Morphological Opening
68 img = cv2.morphologyEx(img, cv2.MORPH_OPEN, Kernel)
69
70
71 if TestMode:
72     plt.figure(figsize=(13, 3))
73     librosa.display.specshow(img, sr=sr, fmin=0, cmap='gray_r',
74                             y_axis='linear', x_axis = 's')
75
76     plt.tick_params( labelleft=False,labelbottom=False)
77     plt.title('Selected pixels after opening', fontsize = 20)
78     plt.ylabel('Frequency [Hz]', fontsize = 18)
79     plt.xlabel('Time [s]', fontsize = 18, position = (0.49,0))
80     plt.show()
81
82
83
84 ### STEP 6: Mask in the spectral domain
85 SignalMask = np.zeros((img.shape[1], 1), 'float32')
86 column_mean = np.mean(img, axis=0, keepdims=True)
87 SignalMask[column_mean.T != 0.] = 1
88
89
90 if TestMode:
91
92     plt.figure(figsize=(13, 3))
93     plt.tick_params(labelleft=False,labelbottom=False)
94     plt.scatter(np.arange(1343),SignalMask, cmap = 'gray_r',
95               s=1, color = '#ff7b54')
96     plt.xticks()
97     plt.title('Selected columns', fontsize = 20)
98     plt.ylabel('Mask values [/]', fontsize = 18)
99     plt.xlabel('Time [s]', fontsize = 18, position = (0.49,0))
100    plt.show()
101
102
103
104 ### STEP 7: Dilation of the mask in the spectral domain
105 SignalMask = ndimage.morphology.binary_dilation(SignalMask,
106         iterations=2).astype(SignalMask.dtype)
107
108
109
110 if TestMode:
111
112     plt.figure(figsize=(13, 3))
113     plt.tick_params(labelleft=False, labelbottom=False)

```

```

114     plt.scatter(np.arange(1343),SignalMask, cmap = 'gray_r',
115                 s =1, color = '#ff7b54')
116     plt.ylabel('Mask values [/]', fontsize = 18)
117     plt.xlabel('Time [s]', fontsize = 18, position = (0.49,0))
118     plt.xticks()
119     plt.title('Selected columns after two passes of dilation', fontsize = 20)
120     plt.show()
121
122
123
124     ## Step 8 : Scale the mask in the temporal domain
125     SignalMaskFinal = np.zeros((len(CompleteAudio), 1))
126     SignalMaskFinal = np.kron(SignalMask, np.ones((int(len(CompleteAudio) /
127             img.shape[1]), 1), dtype=int))
128     SignalMaskFinal = SignalMaskFinal.squeeze()
129     if(len(SignalMaskFinal) != len(CompleteAudio)):
130         ToBeAppended = np.array([SignalMaskFinal[-1]] * (len(CompleteAudio) -
131             len(SignalMaskFinal)))
132         ToBeAppended = ToBeAppended.tolist()
133         SignalMaskFinal = SignalMaskFinal.tolist()
134         SignalMaskFinal.extend(ToBeAppended)
135         SignalMaskFinal = np.array(SignalMaskFinal,dtype=object)
136
137
138
139     if TestMode:
140
141         plt.figure(figsize=(13, 3))
142         plt.tick_params(labelleft=False,labelbottom=False)
143         abs = np.linspace(0, len(CompleteAudio) / sr, num=len(CompleteAudio))
144         abs = np.transpose(np.array([abs]))
145         librosa.display.waveplot(CompleteAudio, sr)
146         plt.plot(abs,SignalMaskFinal*np.max(CompleteAudio)
147             + np.mean(CompleteAudio), color = '#ff7b54')
148         plt.title('Signal Mask', fontsize = 20)
149         plt.ylabel('Amplitude [/]', fontsize = 18)
150         plt.xlabel('Time [s]', fontsize = 18, position = (0.49,0))
151         plt.show()
152
153
154     #Return the signal mask containing only values equal to 0 and 1
155     return SignalMaskFinal

```

A.2 Implementation and illustration of the `step_decay` schedule on *Keras*.

The implementation is straight-forward and given in the following piece of code with some comments:

```
1 import math
2
3 def step_decay(epoch):
4     initial_lrate = 1e-3 #Initial value of the learning rate (to be modified)
5
6     drop = 0.5 #The learning rate will be multiplied by 'drop'
7               #each 'epochs_drop' epochs
8
9     epochs_drop = 10 #Number of epochs to reach to divide the
10                    #learning rate by a factor 'drop'
11
12     lrate = initial_lrate * math.pow(drop, math.floor((1 + epoch) / epochs_drop))
13
14     return lrate
```

The above schedule is used in the *LearningRateScheduler* Keras callback as follow:

```
1 from tensorflow.keras.callbacks import LearningRateScheduler
2 lrate = LearningRateScheduler(step_decay)
```

An illustration of the schedule is given in Figure A.1.

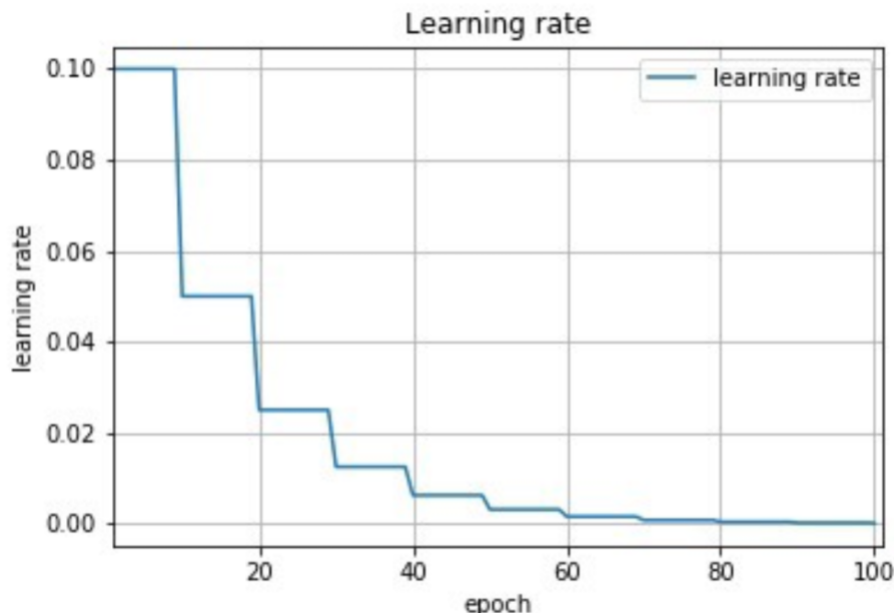


Figure A.1: `Step_decay` schedule used by the *LearningRateScheduler* Keras callback. The learning rate is divided by two every ten epochs in this example [246].

A.3 Implementation of the weighted categorical crossentropy loss on *Keras*

The implementation of the loss in the *Keras* framework, based on [154], is given below.:

```
1 from keras import backend as K
2
3 def weighted_categorical_crossentropy(weights):
4     """
5     A weighted version of keras.objectives.categorical_crossentropy
6
7     Variables:
8         weights: numpy array of shape (C,) where C is the number of classes
9
10    Usage:
11        weights = np.array([0.5,2,10]) # Class one at 0.5, class 2 twice the
12        #normal weights, class 3 10x the normal weights.
13        loss = weighted_categorical_crossentropy(weights)
14        model.compile(loss=loss,optimizer='adam') #optimizer can be modified
15    """
16
17    weights = K.variable(weights)
18
19    def loss(y_true, y_pred):
20        # scale predictions so that the class probas of each sample sum to 1
21        y_pred /= K.sum(y_pred, axis=-1, keepdims=True)
22        # clip to prevent NaN's and Inf's
23        y_pred = K.clip(y_pred, K.epsilon(), 1 - K.epsilon())
24        # calc
25        loss = y_true * K.log(y_pred) * weights
26        loss = -K.sum(loss, -1)
27        return loss
28
29    return loss
```

A.4 Hyperparameters of the seven studied features

We show all the studied variations for each of the seven features. Each table corresponds to the variations of one spectral feature. The feature hyperparameters have been chosen based on the literature.

Name	FFT size	Overlapping size
Spectrogram_0	128	32
Spectrogram_1	128	64
Spectrogram_2	128	96
Spectrogram_3	128	128
Spectrogram_4	256	64
Spectrogram_5	256	128
Spectrogram_6	256	192
Spectrogram_7	256	256
Spectrogram_8	512	128
Spectrogram_9	512	256
Spectrogram_10	512	384

Table A.1: Spectrogram features studied in Section 5.3.2.

Name	FFT size	Overlapping size
LAspectrogram_0	128	32
LAspectrogram_1	128	64
LAspectrogram_2	128	96
LAspectrogram_3	128	128
LAspectrogram_4	256	64
LAspectrogram_5	256	128
LAspectrogram_6	256	192
LAspectrogram_7	256	256
LAspectrogram_8	512	128
LAspectrogram_9	512	256
LAspectrogram_10	512	384

Table A.2: Log-amplitude spectrogram features studied in Section 5.3.2.

Name	FFT size	Overlapping size	Number of mel filters
Melspectrogram_0	128	32	32
Melspectrogram_1	128	64	32
Melspectrogram_2	128	96	32
Melspectrogram_3	128	128	32
Melspectrogram_4	256	64	32
Melspectrogram_5	256	128	32
Melspectrogram_6	256	192	32
Melspectrogram_7	256	256	32
Melspectrogram_8	256	64	64
Melspectrogram_9	256	128	64
Melspectrogram_10	256	192	64
Melspectrogram_11	256	256	64
Melspectrogram_12	512	128	32
Melspectrogram_13	512	256	32
Melspectrogram_14	512	384	32
Melspectrogram_15	512	128	64
Melspectrogram_16	512	256	64
Melspectrogram_17	512	384	64
Melspectrogram_18	512	128	128
Melspectrogram_19	512	256	128
Melspectrogram_20	512	384	128
Melspectrogram_21	1024	256	32
Melspectrogram_22	1024	256	64
Melspectrogram_23	1024	256	128

Table A.3: Mel spectrogram features studied in Section 5.3.2.

Name	FFT size	Overlapping size	Number of mel filters	Number of coefficients
MFCC_0	128	32	32	32
MFCC_1	128	64	32	32
MFCC_2	128	96	32	32
MFCC_3	128	128	32	32
MFCC_4	256	64	32	32
MFCC_5	256	128	32	32
MFCC_6	256	192	32	32
MFCC_7	256	256	32	32
MFCC_8	256	64	64	39
MFCC_9	256	128	64	39
MFCC_10	256	192	64	39
MFCC_11	256	256	64	39
MFCC_12	512	128	32	32
MFCC_13	512	256	32	32
MFCC_14	512	384	32	32
MFCC_15	512	128	64	39
MFCC_16	512	256	64	39
MFCC_17	512	384	64	39
MFCC_18	512	128	128	39
MFCC_19	512	256	128	39
MFCC_20	512	384	128	39
MFCC_21	1024	256	32	32
MFCC_22	1024	256	64	39
MFCC_23	1024	256	128	39

Table A.4: MFCC features studied in Section 5.3.2.

Name	FFT size	Overlapping size	Number of gammatone filters	Coefficients scale
Cochleagram_0	128	32	32	Linear
Cochleagram_1	128	64	32	Linear
Cochleagram_2	128	96	32	Linear
Cochleagram_3	128	128	32	Linear
Cochleagram_4	128	32	64	Linear
Cochleagram_5	128	64	64	Linear
Cochleagram_6	128	96	64	Linear
Cochleagram_7	128	128	64	Linear
Cochleagram_8	128	32	128	Linear
Cochleagram_9	128	64	128	Linear
Cochleagram_10	128	96	128	Linear
Cochleagram_11	128	128	128	Linear
Cochleagram_12	256	192	32	Linear
Cochleagram_13	256	256	32	Linear
Cochleagram_14	256	192	64	Linear
Cochleagram_15	256	256	64	Linear
Cochleagram_16	256	192	128	Linear
Cochleagram_17	256	256	128	Linear
Cochleagram_18	512	384	32	Linear
Cochleagram_19	512	384	64	Linear
Cochleagram_20	512	384	128	Linear
Cochleagram_21	128	32	32	Log
Cochleagram_22	128	64	32	Log
Cochleagram_23	128	96	32	Log
Cochleagram_24	128	128	32	Log
Cochleagram_25	128	32	64	Log
Cochleagram_26	128	64	64	Log
Cochleagram_27	128	96	64	Log
Cochleagram_28	128	128	64	Log
Cochleagram_29	128	32	128	Log
Cochleagram_30	128	64	128	Log
Cochleagram_31	128	96	128	Log
Cochleagram_32	128	128	128	Log
Cochleagram_33	256	192	32	Log
Cochleagram_34	256	256	32	Log
Cochleagram_35	256	192	64	Log
Cochleagram_36	256	256	64	Log
Cochleagram_37	256	192	128	Log
Cochleagram_38	256	256	128	Log
Cochleagram_39	512	384	32	Log
Cochleagram_40	512	384	64	Log
Cochleagram_41	512	384	128	Log

Table A.5: Cochleagram features studied in Section 5.3.2.

Name	FFT size	Overlapping size	Number of gammatone filters	Number of coefficients	Coefficients scale
GFCC_0	128	32	32	32	Linear
GFCC_1	128	64	32	32	Linear
GFCC_2	128	96	32	32	Linear
GFCC_3	128	128	32	32	Linear
GFCC_4	128	32	64	39	Linear
GFCC_5	128	64	64	39	Linear
GFCC_6	128	96	64	39	Linear
GFCC_7	128	128	64	39	Linear
GFCC_8	128	32	128	39	Linear
GFCC_9	128	64	128	39	Linear
GFCC_10	128	96	128	39	Linear
GFCC_11	128	128	128	39	Linear
GFCC_12	256	192	32	32	Linear
GFCC_13	256	256	32	32	Linear
GFCC_14	256	192	64	32	Linear
GFCC_15	256	256	64	39	Linear
GFCC_16	256	192	128	39	Linear
GFCC_17	256	256	128	39	Linear
GFCC_18	512	384	32	32	Linear
GFCC_19	512	384	64	39	Linear
GFCC_20	512	384	128	39	Linear
GFCC_21	128	32	32	32	Log
GFCC_22	128	64	32	32	Log
GFCC_23	128	96	32	32	Log
GFCC_24	128	128	32	32	Log
GFCC_25	128	32	64	39	Log
GFCC_26	128	64	64	39	Log
GFCC_27	128	96	64	39	Log
GFCC_28	128	128	64	39	Log
GFCC_29	128	32	128	39	Log
GFCC_30	128	64	128	39	Log
GFCC_31	128	96	128	39	Log
GFCC_32	128	128	128	39	Log
GFCC_33	256	192	32	32	Log
GFCC_34	256	256	32	32	Log
GFCC_35	256	192	64	32	Log
GFCC_36	256	256	64	39	Log
GFCC_37	256	192	128	39	Log
GFCC_38	256	256	128	39	Log
GFCC_39	512	384	32	32	Log
GFCC_40	512	384	64	39	Log
GFCC_41	512	384	128	39	Log

Table A.6: GFCC features studied in Section 5.3.2.

Name	Wavelet	Scale	Number of temporal frames
Scalogram_0	"morl"	33	497
Scalogram_1	"morl"	33	249
Scalogram_2	"morl"	33	166
Scalogram_3	"morl"	33	125
Scalogram_4	"morl"	33	83
Scalogram_5	"morl"	33	62
Scalogram_6	"morl"	33	41
Scalogram_7	"morl"	33	59
Scalogram_8	"morl"	65	497
Scalogram_9	"morl"	65	249
Scalogram_10	"morl"	65	166
Scalogram_11	"morl"	65	125
Scalogram_12	"morl"	65	83
Scalogram_13	"morl"	65	62
Scalogram_14	"morl"	65	41
Scalogram_15	"morl"	65	59
Scalogram_16	"morl"	129	497
Scalogram_17	"morl"	129	249
Scalogram_18	"morl"	129	166
Scalogram_19	"morl"	129	125
Scalogram_20	"morl"	129	83
Scalogram_21	"morl"	129	62
Scalogram_22	"morl"	129	41
Scalogram_23	"morl"	129	59

Table A.7: Scalogram features studied in Section 5.3.2.

A.5 CMSIS-NN implementation files

We present hereafter the four files required for a clean implementation in *CMSIS-NN*. The parameters, weights, and input image depend on the application and the network. We show the ones obtained with our refined optimized model. Furthermore, the proposed main code performs, in that case, a for loop of 100 different inferences. To that purpose, the input.h code has a buffer containing 100 images flattened line by line. Finally, note that the presented main file does not consider all optimizations from the microcontroller in Section 7.4, such that it remains clear.

A.5.1 main.cpp

```
1  #include <stdint.h>
2  #include <stdio.h>
3  #include "arm_math.h"
4  #include "NN_parameter.h"
5  #include "NN_weights.h"
6  #include "arm_nnfunctions.h"
7  #include "NN_inputs.h"
8
9
10 #ifdef _RTE_
11 #include "RTE_Components.h"
12 #ifdef RTE_Compiler_EventRecorder
13 #include "EventRecorder.h"
14 #endif
15 #endif
16
17
18 // input and weights
19
20 static q7_t conv1_ds_wt[DepthwiseCONV1_IM_CH * DepthwiseCONV1_KER_DIM *
21     DepthwiseCONV1_KER_DIM * DepthwiseCONV1_OUT_CH] = CONV1_DS_WT;
22 static q7_t conv2_ds_wt[DepthwiseCONV2_IM_CH * DepthwiseCONV2_KER_DIM *
23     DepthwiseCONV2_KER_DIM * DepthwiseCONV2_OUT_CH] = CONV2_DS_WT;
24 static q7_t conv3_ds_wt[DepthwiseCONV3_IM_CH * DepthwiseCONV3_KER_DIM *
25     DepthwiseCONV3_KER_DIM * DepthwiseCONV3_OUT_CH] = CONV3_DS_WT;
26 static q7_t conv4_ds_wt[DepthwiseCONV4_IM_CH * DepthwiseCONV4_KER_DIM *
27     DepthwiseCONV4_KER_DIM * DepthwiseCONV4_OUT_CH] = CONV4_DS_WT;
28
29
30 static q7_t conv1_pw_wt[PointwiseCONV1_IM_CH * PointwiseCONV1_KER_DIM *
31     PointwiseCONV1_KER_DIM * PointwiseCONV1_OUT_CH] = CONV1_PW_WT;
32 static q7_t conv2_pw_wt[PointwiseCONV2_IM_CH * PointwiseCONV2_KER_DIM *
33     PointwiseCONV2_KER_DIM * PointwiseCONV2_OUT_CH] = CONV2_PW_WT;
34 static q7_t conv3_pw_wt[PointwiseCONV3_IM_CH * PointwiseCONV3_KER_DIM *
35     PointwiseCONV3_KER_DIM * PointwiseCONV3_OUT_CH] = CONV3_PW_WT;
36 static q7_t conv4_pw_wt[PointwiseCONV4_IM_CH * PointwiseCONV4_KER_DIM *
37     PointwiseCONV4_KER_DIM * PointwiseCONV4_OUT_CH] = CONV4_PW_WT;
38
39 static q7_t conv1_ds_bias[DepthwiseCONV1_OUT_CH] = CONV1_DS_BIAS;
```

```

40 static q7_t conv2_ds_bias[DepthwiseCONV2_OUT_CH] = CONV2_DS_BIAS;
41 static q7_t conv3_ds_bias[DepthwiseCONV3_OUT_CH] = CONV3_DS_BIAS;
42 static q7_t conv4_ds_bias[DepthwiseCONV4_OUT_CH] = CONV4_DS_BIAS;
43
44
45 static q7_t conv1_pw_bias[PointwiseCONV1_OUT_CH] = CONV1_PW_BIAS;
46 static q7_t conv2_pw_bias[PointwiseCONV2_OUT_CH] = CONV2_PW_BIAS;
47 static q7_t conv3_pw_bias[PointwiseCONV3_OUT_CH] = CONV3_PW_BIAS;
48 static q7_t conv4_pw_bias[PointwiseCONV4_OUT_CH] = CONV4_PW_BIAS;
49
50
51 static q7_t ip1_wt[IP1_DIM * IP1_OUT] = FINAL_FC_WT;
52 static q7_t ip1_bias[IP1_OUT] = FINAL_FC_BIAS;
53
54
55
56 int8_t image_dataRuntimeExecution[DepthwiseCONV1_IM_CH * DepthwiseCONV1_IN_DIMX
57 * DepthwiseCONV1_IN_DIMY*100] = IMG_DATA_RUNTIMEEXECUTION;
58
59
60 q7_t output_data[IP1_OUT];
61
62 q7_t col_buffer[2 * 3 * 3 * 40 * 2];
63
64 q7_t scratch_buffer[20*8*8 + 10*4*8] ;
65
66
67
68
69 int main()
70 {
71
72 #ifdef RTE_Compiler_EventRecorder
73 EventRecorderInitialize (EventRecordAll, 1);
74 #endif
75
76
77
78 //printf("start execution\n");
79 /* start the execution */
80
81
82
83 q7_t *img_buffer1 = scratch_buffer;
84
85
86 q7_t *img_buffer2 = img_buffer1 + 10*4*8 ;
87
88
89

```

```

90
91  for (int j=0;j<100; j+=1) {
92
93
94  /* input pre-processing */
95
96  for (int i=0;i<39*15*1; i+=1) {
97      img_buffer2[i] = (q7_t) image_dataRuntimeExecution[j*585 + i] ;
98  }
99
100
101
102  //Network inference
103  // Layer 1
104
105  arm_depthwise_separable_conv_HWC_q7_nonsquare(img_buffer2, DepthwiseCONV1_IN_DIMX,
106  DepthwiseCONV1_IN_DIMY, DepthwiseCONV1_IM_CH, conv1_ds_wt, DepthwiseCONV1_OUT_CH,
107  DepthwiseCONV1_KER_DIM,DepthwiseCONV1_KER_DIM, DepthwiseCONV1_PADDINGX,
108  DepthwiseCONV1_PADDINGX, DepthwiseCONV1_STRIDE, DepthwiseCONV1_STRIDE,
109  conv1_ds_bias, CONV1_DS_BIAS_LSHIFT, CONV1_DS_OUT_RSHIFT, img_buffer1,
110  DepthwiseCONV1_OUT_DIMX,DepthwiseCONV1_OUT_DIMY, (q15_t *) col_buffer, NULL);
111
112
113
114  arm_convolve_HWC_q7_basic_nonsquare(img_buffer1, PointwiseCONV1_IN_DIMX,
115  PointwiseCONV1_IN_DIMY, PointwiseCONV1_IM_CH, conv1_pw_wt, PointwiseCONV1_OUT_CH,
116  PointwiseCONV1_KER_DIM, PointwiseCONV1_KER_DIM, PointwiseCONV1_PADDINGX,
117  PointwiseCONV1_PADDINGX, PointwiseCONV1_STRIDE, PointwiseCONV1_STRIDE,
118  conv1_pw_bias, CONV1_PW_BIAS_LSHIFT, CONV1_PW_OUT_RSHIFT, img_buffer2,
119  PointwiseCONV1_OUT_DIMX,PointwiseCONV1_OUT_DIMY, (q15_t *) col_buffer, NULL);
120
121
122
123  arm_relu_q7(img_buffer2, PointwiseCONV1_OUT_DIMX * PointwiseCONV1_OUT_DIMY *
124  PointwiseCONV1_OUT_CH);
125
126
127
128  // Layer 2
129
130  arm_depthwise_separable_conv_HWC_q7_nonsquare(img_buffer2, DepthwiseCONV2_IN_DIMX,
131  DepthwiseCONV2_IN_DIMY, DepthwiseCONV2_IM_CH, conv2_ds_wt, DepthwiseCONV2_OUT_CH,
132  DepthwiseCONV2_KER_DIM,DepthwiseCONV2_KER_DIM, DepthwiseCONV2_PADDINGX,
133  DepthwiseCONV2_PADDINGX, DepthwiseCONV2_STRIDE, DepthwiseCONV2_STRIDE,
134  conv2_ds_bias, CONV2_DS_BIAS_LSHIFT, CONV2_DS_OUT_RSHIFT, img_buffer1,
135  DepthwiseCONV2_OUT_DIMX,DepthwiseCONV2_OUT_DIMY, (q15_t *) col_buffer, NULL);
136
137
138
139  arm_convolve_1x1_HWC_q7_fast_nonsquare(img_buffer1, PointwiseCONV2_IN_DIMX,

```

```

140 PointwiseCONV2_IN_DIMY, PointwiseCONV2_IM_CH, conv2_pw_wt, PointwiseCONV2_OUT_CH,
141 PointwiseCONV2_KER_DIM, PointwiseCONV2_KER_DIM, PointwiseCONV2_PADDINGX,
142 PointwiseCONV2_PADDINGX, PointwiseCONV2_STRIDE, PointwiseCONV2_STRIDE,
143 conv2_pw_bias, CONV2_PW_BIAS_LSHIFT, CONV2_PW_OUT_RSHIFT, img_buffer2,
144 PointwiseCONV2_OUT_DIMX, PointwiseCONV2_OUT_DIMY, (q15_t *) col_buffer, NULL);
145
146
147
148 arm_relu_q7(img_buffer2, PointwiseCONV2_OUT_DIMX * PointwiseCONV2_OUT_DIMY *
149 PointwiseCONV2_OUT_CH);
150
151
152
153 // Layer 3
154
155
156 arm_depthwise_separable_conv_HWC_q7_nonsquare(img_buffer2, DepthwiseCONV3_IN_DIMX,
157 DepthwiseCONV3_IN_DIMY, DepthwiseCONV3_IM_CH, conv3_ds_wt, DepthwiseCONV3_OUT_CH,
158 DepthwiseCONV3_KER_DIM, DepthwiseCONV3_KER_DIM, DepthwiseCONV3_PADDINGX,
159 DepthwiseCONV3_PADDINGX, DepthwiseCONV3_STRIDE, DepthwiseCONV3_STRIDE,
160 conv3_ds_bias, CONV3_DS_BIAS_LSHIFT, CONV3_DS_OUT_RSHIFT, img_buffer1,
161 DepthwiseCONV3_OUT_DIMX, DepthwiseCONV3_OUT_DIMY, (q15_t *) col_buffer, NULL);
162
163
164
165 arm_convolve_1x1_HWC_q7_fast_nonsquare(img_buffer1, PointwiseCONV3_IN_DIMX,
166 PointwiseCONV3_IN_DIMY, PointwiseCONV3_IM_CH, conv3_pw_wt, PointwiseCONV3_OUT_CH,
167 PointwiseCONV3_KER_DIM, PointwiseCONV3_KER_DIM, PointwiseCONV3_PADDINGX,
168 PointwiseCONV3_PADDINGX, PointwiseCONV3_STRIDE, PointwiseCONV3_STRIDE,
169 conv3_pw_bias, CONV3_PW_BIAS_LSHIFT, CONV3_PW_OUT_RSHIFT, img_buffer2,
170 PointwiseCONV3_OUT_DIMX, PointwiseCONV3_OUT_DIMY, (q15_t *) col_buffer, NULL);
171
172
173
174 arm_relu_q7(img_buffer2, PointwiseCONV3_OUT_DIMX * PointwiseCONV3_OUT_DIMY *
175 PointwiseCONV3_OUT_CH);
176
177
178
179 // Layer 4
180
181
182 arm_depthwise_separable_conv_HWC_q7_nonsquare(img_buffer2, DepthwiseCONV4_IN_DIMX,
183 DepthwiseCONV4_IN_DIMY, DepthwiseCONV4_IM_CH, conv4_ds_wt, DepthwiseCONV4_OUT_CH,
184 DepthwiseCONV4_KER_DIM, DepthwiseCONV4_KER_DIM, DepthwiseCONV4_PADDINGX,
185 DepthwiseCONV4_PADDINGX, DepthwiseCONV4_STRIDE, DepthwiseCONV4_STRIDE,
186 conv4_ds_bias, CONV4_DS_BIAS_LSHIFT, CONV4_DS_OUT_RSHIFT, img_buffer1,
187 DepthwiseCONV4_OUT_DIMX, DepthwiseCONV4_OUT_DIMY, (q15_t *) col_buffer, NULL);
188
189

```

```

190
191
192 arm_convolve_1x1_HWC_q7_fast_nonsquare(img_buffer1, PointwiseCONV4_IN_DIMX,
193 PointwiseCONV4_IN_DIMY, PointwiseCONV4_IM_CH, conv4_pw_wt, PointwiseCONV4_OUT_CH,
194 PointwiseCONV4_KER_DIM,PointwiseCONV4_KER_DIM, PointwiseCONV4_PADDINGX,
195 PointwiseCONV4_PADDINGX, PointwiseCONV4_STRIDE, PointwiseCONV4_STRIDE,
196 conv4_pw_bias, CONV4_PW_BIAS_LSHIFT, CONV4_PW_OUT_RSHIFT, img_buffer2,
197 PointwiseCONV4_OUT_DIMX,PointwiseCONV4_OUT_DIMY, (q15_t *) col_buffer, NULL);
198
199
200
201 arm_relu_q7(img_buffer2, PointwiseCONV4_OUT_DIMX * PointwiseCONV4_OUT_DIMY *
202 PointwiseCONV4_OUT_CH);
203
204
205
206 // Layer 5
207
208
209 arm_fully_connected_q7_opt(img_buffer2, ip1_wt, IP1_DIM, IP1_OUT,
210 FINAL_FC_BIAS_LSHIFT, FINAL_FC_OUT_RSHIFT, ip1_bias, output_data,
211 (q15_t *)img_buffer1 );
212
213
214
215 for (int i = 0; i < 4; i++)
216 {
217     printf("%d: %d\n", i, output_data[i]);
218 }
219
220
221
222 arm_softmax_q7(output_data, 4, output_data);
223
224
225
226 for (int i = 0; i < 4; i++)
227 {
228     printf("%d: %d\n", i, output_data[i]);
229 }
230
231
232 }
233
234 //printf("end of execution\n");
235
236 return 0;
237
238 }

```

A.5.2 parameter.h

```
1 //First depthwise convolution
2 #define DepthwiseCONV1_OUT_CH 1
3 #define DepthwiseCONV1_IM_CH 1
4 #define DepthwiseCONV1_IN_DIMX 15
5 #define DepthwiseCONV1_IN_DIMY 39
6 #define DepthwiseCONV1_KER_DIM 3
7 #define DepthwiseCONV1_PADDINGX 1
8 #define DepthwiseCONV1_PADDINGX 1
9 #define DepthwiseCONV1_STRIDE 2
10 #define DepthwiseCONV1_OUT_DIMX 8
11 #define DepthwiseCONV1_OUT_DIMY 20
12
13
14 //First pointwise convolution
15 #define PointwiseCONV1_OUT_CH 8
16 #define PointwiseCONV1_IM_CH 1
17 #define PointwiseCONV1_IN_DIMX 8
18 #define PointwiseCONV1_IN_DIMY 20
19 #define PointwiseCONV1_KER_DIM 1
20 #define PointwiseCONV1_PADDINGX 0
21 #define PointwiseCONV1_PADDINGX 0
22 #define PointwiseCONV1_STRIDE 1
23 #define PointwiseCONV1_OUT_DIMX 8
24 #define PointwiseCONV1_OUT_DIMY 20
25
26
27 //Second depthwise convolution
28 #define DepthwiseCONV2_OUT_CH 8
29 #define DepthwiseCONV2_IM_CH 8
30 #define DepthwiseCONV2_IN_DIMX 8
31 #define DepthwiseCONV2_IN_DIMY 20
32 #define DepthwiseCONV2_KER_DIM 3
33 #define DepthwiseCONV2_PADDINGX 0
34 #define DepthwiseCONV2_PADDINGX 0
35 #define DepthwiseCONV2_STRIDE 2
36 #define DepthwiseCONV2_OUT_DIMX 4
37 #define DepthwiseCONV2_OUT_DIMY 10
38
39
40 //Second pointwise convolution
41 #define PointwiseCONV2_OUT_CH 8
42 #define PointwiseCONV2_IM_CH 8
43 #define PointwiseCONV2_IN_DIMX 4
44 #define PointwiseCONV2_IN_DIMY 10
45 #define PointwiseCONV2_KER_DIM 1
46 #define PointwiseCONV2_PADDINGX 0
47 #define PointwiseCONV2_PADDINGX 0
48 #define PointwiseCONV2_STRIDE 1
```

```

49 #define PointwiseCONV2_OUT_DIMX 4
50 #define PointwiseCONV2_OUT_DIMY 10
51
52
53 //Third depthwise convolution
54 #define DepthwiseCONV3_OUT_CH 8
55 #define DepthwiseCONV3_IM_CH 8
56 #define DepthwiseCONV3_IN_DIMX 4
57 #define DepthwiseCONV3_IN_DIMY 10
58 #define DepthwiseCONV3_KER_DIM 3
59 #define DepthwiseCONV3_PADDINGX 0
60 #define DepthwiseCONV3_PADDINGX 0
61 #define DepthwiseCONV3_STRIDE 2
62 #define DepthwiseCONV3_OUT_DIMX 2
63 #define DepthwiseCONV3_OUT_DIMY 5
64
65
66 //Third pointwise convolution
67 #define PointwiseCONV3_OUT_CH 16
68 #define PointwiseCONV3_IM_CH 8
69 #define PointwiseCONV3_IN_DIMX 2
70 #define PointwiseCONV3_IN_DIMY 5
71 #define PointwiseCONV3_KER_DIM 1
72 #define PointwiseCONV3_PADDINGX 0
73 #define PointwiseCONV3_PADDINGX 0
74 #define PointwiseCONV3_STRIDE 1
75 #define PointwiseCONV3_OUT_DIMX 2
76 #define PointwiseCONV3_OUT_DIMY 5
77
78
79 //Last depthwise convolution
80 #define DepthwiseCONV4_OUT_CH 16
81 #define DepthwiseCONV4_IM_CH 16
82 #define DepthwiseCONV4_IN_DIMX 2
83 #define DepthwiseCONV4_IN_DIMY 5
84 #define DepthwiseCONV4_KER_DIM 3
85 #define DepthwiseCONV4_PADDINGX 0
86 #define DepthwiseCONV4_PADDINGX 1
87 #define DepthwiseCONV4_STRIDE 2
88 #define DepthwiseCONV4_OUT_DIMX 1
89 #define DepthwiseCONV4_OUT_DIMY 3
90
91
92 //Last pointwise convolution
93 #define PointwiseCONV4_OUT_CH 40
94 #define PointwiseCONV4_IM_CH 16
95 #define PointwiseCONV4_IN_DIMX 1
96 #define PointwiseCONV4_IN_DIMY 3
97 #define PointwiseCONV4_KER_DIM 1
98 #define PointwiseCONV4_PADDINGX 0

```

```

99 #define PointwiseCONV4_PADDING 0
100 #define PointwiseCONV4_STRIDE 1
101 #define PointwiseCONV4_OUT_DIMX 1
102 #define PointwiseCONV4_OUT_DIMY 3
103
104
105 //Fully Connected Layer
106 #define IP1_DIM 40*3*1 // 120
107 #define IP1_OUT 4

```

A.5.3 weights.h

```

1 #define CONV1_DS_WT {50,-26,-33,80,-12,-121,68,23,20}
2 #define CONV1_DS_BIAS {-32}
3 #define CONV1_PW_WT {50,62,17,-51,-71,-33,-84,46}
4 #define CONV1_PW_BIAS {-2,-77,64,3,-29,40,60,40}
5 #define CONV2_DS_WT {37,48,-81,-41,94,79,0,4,89,71,20,63,56,116,-83,11,-7,-8,...}
6 #define CONV2_PW_WT {28,-11,-38,39,0,36,21,-24,-31,53,-20,-18,-10,17,0,52,71,...}
7 #define CONV2_PW_BIAS {-26,75,23,-11,28,19,37,101}
8 #define CONV3_DS_WT {40,-13,51,-29,-47,-21,73,-52,32,7,108,-80,15,-27,108,-70,...}
9 #define CONV3_PW_WT {31,-17,-51,12,-48,-14,-6,46,-24,3,8,17,8,-1,-34,11,6,24,...}
10 #define CONV3_PW_BIAS {44,-128,30,-4,-35,-73,-83,12,-88,109,60,112,52,-26,-87,68}
11 #define CONV4_DS_WT {-1,-14,105,-43,-20,-32,-8,-95,-37,70,-10,57,39,52,-57,...}
12 #define CONV4_DS_BIAS {-10,-6,29,40,11,-28,8,-16,-31,-75,-37,40,-5,-2,-7,20}
13 #define CONV4_PW_WT {54,29,-20,-11,84,-11,20,-77,88,-88,28,49,25,29,-15,-35,...}
14 #define CONV4_PW_BIAS {27,-9,51,41,-19,-105,30,5,-6,0,-60,-5,-1,-12,-11,72,...}
15 #define FINAL_FC_WT {8,6,9,14,6,-17,-7,1,-12,14,6,3,-2,2,-3,2,-10,14,11,5,3,...}
16 #define FINAL_FC_BIAS {50,-75,9,-4}
17
18
19
20 // Bias and output shifts
21 #define CONV1_DS_BIAS_LSHIFT 4
22 #define CONV1_DS_OUT_RSHIFT 6
23 #define CONV1_PW_BIAS_LSHIFT 4
24 #define CONV1_PW_OUT_RSHIFT 6
25
26
27 #define CONV2_DS_BIAS_LSHIFT 3
28 #define CONV2_DS_OUT_RSHIFT 7
29 #define CONV2_PW_BIAS_LSHIFT 5
30 #define CONV2_PW_OUT_RSHIFT 6
31
32
33
34 #define CONV3_DS_BIAS_LSHIFT 3
35 #define CONV3_DS_OUT_RSHIFT 7
36 #define CONV3_PW_BIAS_LSHIFT 4

```

```

37 #define CONV3_PW_OUT_RSHIFT 6
38
39
40
41 #define CONV4_DS_BIAS_LSHIFT 3
42 #define CONV4_DS_OUT_RSHIFT 6
43 #define CONV4_PW_BIAS_LSHIFT 6
44 #define CONV4_PW_OUT_RSHIFT 6
45
46
47 #define FINAL_FC_BIAS_LSHIFT 4
48 #define FINAL_FC_OUT_RSHIFT 8

```

A.5.4 input.h

```

1 #define IMG_DATA_RUNTIMEEXECUTION {1,2,3,3,2,2,2,2,2,1,1,1,1,1,1,1,2,3,3,2,2,...}

```

- [1] T. Pellegrini, "Densely connected cnns for bird audio detection," in *2017 25th European Signal Processing Conference (EUSIPCO)*, pp. 1734–1738, IEEE, 2017.
- [2] S. Hershey, S. Chaudhuri, D. P. Ellis, J. F. Gemmeke, A. Jansen, R. C. Moore, M. Plakal, D. Platt, R. A. Saurous, B. Seybold, *et al.*, "Cnn architectures for large-scale audio classification," in *2017 IEEE International Conference On Acoustics, Speech and Signal Processing (ICASSP)*, pp. 131–135, IEEE, 2017.
- [3] Y. M. Costa, L. S. Oliveira, and C. N. Silla Jr, "An evaluation of convolutional neural networks for music classification using spectrograms," *Applied soft computing*, vol. 52, pp. 28–38, 2017.
- [4] N. D. Lane, S. Bhattacharya, A. Mathur, P. Georgiev, C. Forlivesi, and F. Kawsar, "Squeezing deep learning into mobile and embedded devices," *IEEE Pervasive Computing*, vol. 16, no. 3, pp. 82–88, 2017.
- [5] P. Warden and D. Situnayake, *Tinyml: Machine learning with tensorflow lite on arduino and ultra-low-power microcontrollers*. " O'Reilly Media, Inc.", 2019.
- [6] Y. Zhang, N. Suda, L. Lai, and V. Chandra, "Hello edge: Keyword spotting on microcontrollers," *arXiv preprint arXiv:1711.07128*, 2017.
- [7] S. Segarceanu, E. Olteanu, and G. Suciu, "Forest monitoring using forest sound identification," in *2020 43rd International Conference on Telecommunications and Signal Processing (TSP)*, pp. 346–349, IEEE, 2020.
- [8] M. Chaudhary, V. Prakash, and N. Kumari, "Identification vehicle movement detection in forest area using mfcc and knn," in *2018 International Conference on System Modeling & Advancement in Research Trends (SMART)*, pp. 158–164, IEEE, 2018.
- [9] P. Kalhara, V. Jayasinghearachchi, A. Dias, V. Ratnayake, C. Jayawardena, and N. Kuruwitaarachchi, "Treespirit: Illegal logging detection and alerting system using audio identification over an iot network," in *2017 11th International Conference on Software, Knowledge, Information Management and Applications (SKIMA)*, pp. 1–7, IEEE, 2017.
- [10] RainForest-Alliance, "9 rainforest facts everyone should know." <https://www.rainforest-alliance.org/articles/9-rainforest-facts-everyone-should-know>, June 2019. Accessed: 2021-02-21.
- [11] R. A. Butler, "10 rainforest facts for 2020." <https://rainforests.mongabay.com/facts/rainforest-facts.html#7>, July 2020. Accessed: 2021-02-21.

- [12] K. Meyer-Schulz and R. Bürger-Arndt, “Les effets de la forêt sur la santé physique et mentale. une revue de la littérature scientifique,” *Sante Publique*, vol. 1, no. HS1, pp. 115–134, 2019.
- [13] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, “Mobilenets: Efficient convolutional neural networks for mobile vision applications,” *arXiv preprint arXiv:1704.04861*, 2017.
- [14] L. Lai, N. Suda, and V. Chandra, “Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus,” *arXiv preprint arXiv:1801.06601*, 2018.
- [15] L. Goldapple, “Saving the rainforests with ai.” <https://atlasofthefuture.org/project/rainforest-connection/>, Feb. 2021. Accessed: 2021-02-18.
- [16] J. Hall, “Poaching animals, explained.” <https://www.nationalgeographic.com/animals/article/poaching-animals/>, Feb. 2019. Accessed: 2021-05-29.
- [17] National Geographic Society, “Wildlife conservation.” <https://www.nationalgeographic.org/encyclopedia/wildlife-conservation/>, Mar. 2020. Accessed: 2021-05-29.
- [18] B. McComb, B. Zuckerberg, D. Vesely, and C. Jordan, *Monitoring animal populations and their habitats: a practitioner’s guide*. CRC Press, 2010.
- [19] M. Rimol, “Understand 3 key types of machine learning.” <https://www.gartner.com/smarterwithgartner/understand-3-key-types-of-machine-learning/>, Mar. 2018. Accessed: 2021-06-02.
- [20] V. Roman, “Machine learning project: Predicting boston house prices with regression.” <https://towardsdatascience.com/machine-learning-project-predicting-boston-house-prices-with-regression-b4e47493633d>, Jan. 2019. Accessed: 2021-06-02.
- [21] ImageNet, “Imagenet large scale visual recognition challenge (ilsvrc).” <https://www.image-net.org/challenges/LSVRC/index.php>. Accessed: 2021-06-02.
- [22] Y. Liang, M. C. Machado, E. Talvitie, and M. Bowling, “State of the art control of atari games using shallow reinforcement learning,” *arXiv preprint arXiv:1512.01563*, 2015.
- [23] A. Prokopov, “Machine learning vs deep learning – when do you need an expert?.” <https://blog.mooncascade.com/machine-learning-vs-deep-learning-when-do-you-need-an-expert/>, Mar. 2019. Accessed: 2021-06-02.
- [24] 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.
- [25] C. Herrmann, M. Ruf, and J. Beyerer, “Cnn-based thermal infrared person detection by domain adaptation,” in *Autonomous Systems: Sensors, Vehicles, Security, and the Internet of Everything*, vol. 10643, p. 1064308, International Society for Optics and Photonics, 2018.
- [26] M. Hussain, J. J. Bird, and D. R. Faria, “A study on cnn transfer learning for image classification,” in *UK Workshop on computational Intelligence*, pp. 191–202, Springer, 2018.
- [27] O. Ronneberger, P. Fischer, and T. Brox, “U-net: Convolutional networks for biomedical image segmentation,” in *International Conference on Medical image computing and computer-assisted intervention*, pp. 234–241, Springer, 2015.
- [28] 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, no. 4, pp. 55–65, 2017.

- [29] T. Williams, R. Li, *et al.*, “An ensemble of convolutional neural networks using wavelets for image classification,” *Journal of Software Engineering and Applications*, vol. 11, no. 02, p. 69, 2018.
- [30] J. O. Nordby, “Environmental sound classification on microcontrollers using convolutional neural networks,” Master’s thesis, Norwegian University of Life Sciences, Ås, 2019.
- [31] S. Pokhrel, “Beginners guide to convolutional neural networks.” <https://towardsdatascience.com/beginners-guide-to-understanding-convolutional-neural-networks-ae9ed58bb17d>, Sep. 2019. Accessed: 2021-06-12.
- [32] mayank2498, “Depth wise separable convolutional neural networks.” <https://www.geeksforgeeks.org/depth-wise-separable-convolutional-neural-networks/>, Aug. 2019. Accessed: 2021-06-13.
- [33] S. Shankar, “What is max pooling in convolutional neural networks?.” <https://www.quora.com/What-is-max-pooling-in-convolutional-neural-networks>, Aug. 2017. Accessed: 2021-06-13.
- [34] J. Brownlee, “How to fix the vanishing gradients problem using the relu.” <https://machinelearningmastery.com/how-to-fix-vanishing-gradients-using-the-rectified-linear-activation-function/>, Jan. 2019. Accessed: 2021-06-13.
- [35] I. Dabbura, “Coding neural network — parameters’ initialization.” <https://mlfromscratch.com/neural-networks-explained/#/>, Apr. 2018. Accessed: 2021-06-02.
- [36] A. Kost, “Applying neural networks for tire pressure monitoring systems,” 2018.
- [37] I. Dabbura, “Coding neural network — parameters’ initialization.” <https://towardsdatascience.com/coding-neural-network-parameters-initialization-f7c2d770e874>, Apr. 2018. Accessed: 2021-06-02.
- [38] S. Nguyen, “A gentle explanation of backpropagation in convolutional neural network (cnn).” <https://medium.com/@ngocson2vn/a-gentle-explanation-of-backpropagation-in-convolutional-neural-network-cnn-1a70abff>, Feb. 2020. Accessed: 2021-06-02.
- [39] Guru99, “Back propagation neural network: What is backpropagation algorithm in machine learning?.” <https://www.guru99.com/backpropagation-neural-network.html>. Accessed: 2021-06-02.
- [40] G. Montavon, G. Orr, and K.-R. Müller, “Neural networks-tricks of the trade second edition,” *Springer, DOI*, vol. 10, pp. 978–3, 2012.
- [41] J. Mohajon, “Confusion matrix for your multi-class machine learning model.” <https://towardsdatascience.com/confusion-matrix-for-your-multi-class-machine-learning-model-ff9aa3bf7826>, May. 2020. Accessed: 2021-06-02.
- [42] M. Hollemans, “How fast is my model?.” <https://machinethink.net/blog/how-fast-is-my-model/>, June 2018. Accessed: 2021-06-02.
- [43] tokusumi, “keras-flops 0.1.2.” <https://pypi.org/project/keras-flops/>. Accessed: 2021-03-13.
- [44] M. Harmer, “Fish classification – cnn model.” <https://mattharmer.me/2020/03/26/fish-classification-cnn-model/>. Accessed: 2021-06-02.

- [45] A. Takimoglu, “What is data augmentation? techniques, benefit and examples.” <https://research.aimultiple.com/data-augmentation/>, Apr. 2021. Accessed: 2021-06-02.
- [46] K. Doshi, “Audio deep learning made simple (part 3): Data preparation and augmentation.” <https://towardsdatascience.com/audio-deep-learning-made-simple-part-3-data-preparation-and-augmentation-24c6e1f6b52> Feb. 2021. Accessed: 2021-06-02.
- [47] K. Xu, D. Feng, H. Mi, B. Zhu, D. Wang, L. Zhang, H. Cai, and S. Liu, “Mixup-based acoustic scene classification using multi-channel convolutional neural network,” in *Pacific Rim conference on multimedia*, pp. 14–23, Springer, 2018.
- [48] S. Wei, K. Xu, D. Wang, F. Liao, H. Wang, and Q. Kong, “Sample mixed-based data augmentation for domestic audio tagging,” *arXiv preprint arXiv:1808.03883*, 2018.
- [49] D. S. Park, W. Chan, Y. Zhang, C.-C. Chiu, B. Zoph, E. D. Cubuk, and Q. V. Le, “SpecAugment: A simple data augmentation method for automatic speech recognition,” *arXiv preprint arXiv:1904.08779*, 2019.
- [50] A. Canziani, “Overfitting and regularization.” <https://atcold.github.io/pytorch-Deep-Learning/en/week14/14-3/>, May 2020. Accessed: 2021-06-02.
- [51] J. Verecken, D. Bol, and C. De Vleeschouwer, *Embedded real-time inference in spiking neural networks for neuromorphic IoT vision sensors*. PhD thesis, Master Thesis. UCL-Ecole polytechnique de Louvain, 2020.
- [52] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: a simple way to prevent neural networks from overfitting,” *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.
- [53] S. Saxena, “Introduction to batch normalization.” <https://www.analyticsvidhya.com/blog/2021/03/introduction-to-batch-normalization/>, Mar. 2021. Accessed: 2021-06-02.
- [54] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” in *International conference on machine learning*, pp. 448–456, PMLR, 2015.
- [55] R. Alake, “Batch normalization in neural networks explained (algorithm breakdown).” <https://towardsdatascience.com/batch-normalization-explained-algorithm-breakdown-23d2794511c>, Apr. 2020. Accessed: 2021-06-02.
- [56] Attune, “Let’s talk sound frequencies!” <https://www.attune.com.au/2020/02/12/lets-talk-sound-frequencies/>. Accessed: 2021-06-02.
- [57] Wikipedia, “Hearing range.” https://en.wikipedia.org/wiki/Hearing_range, June 2021. Accessed: 2021-06-13.
- [58] T. Roelands, “What is a spectrogram?” <https://tomroelandts.com/articles/what-is-a-spectrogram>, Nov. 2013. Accessed: 2021-01-26.
- [59] Wikipedia, “Short-time fourier transform.” https://en.wikipedia.org/wiki/Short-time_Fourier_transform, Dec. 2020. Accessed: 2021-06-13.
- [60] University of Huddersfield, “The short time fourier transform.” <https://learn.flucoma.org/algorithms/stft/>. Accessed: 2021-06-13.

- [61] G. Ananthakrishnan, “Music and speech analysis using the ‘bach’ scale filter-bank,” *M. Sc (Engg) thesis, Indian Institute of Science*, 2007.
- [62] K. J. Piczak, “Esc: Dataset for environmental sound classification,” in *Proceedings of the 23rd ACM international conference on Multimedia*, pp. 1015–1018, 2015.
- [63] F. Font, G. Roma, and X. Serra, “Freesound technical demo,” in *Proceedings of the 21st ACM international conference on Multimedia*, pp. 411–412, 2013.
- [64] K. J. Piczak, “Esc-50 [github repository].” <https://github.com/karolpiczak/ESC-50>, Apr. 2015.
- [65] J. Salamon, C. Jacoby, and J. P. Bello, “A dataset and taxonomy for urban sound research,” in *Proceedings of the 22nd ACM international conference on Multimedia*, pp. 1041–1044, 2014.
- [66] S. Chu, S. Narayanan, and C.-C. J. Kuo, “Environmental sound recognition with time–frequency audio features,” *IEEE Transactions on Audio, Speech, and Language Processing*, vol. 17, no. 6, pp. 1142–1158, 2009.
- [67] M. Cartwright, J. Cramer, A. E. M. Mendez, Y. Wang, H.-H. Wu, V. Lostanlen, M. Fuentes, G. Dove, C. Mydlarz, J. Salamon, *et al.*, “Sonyc-ust-v2: An urban sound tagging dataset with spatiotemporal context,” *arXiv preprint arXiv:2009.05188*, 2020.
- [68] “Dcase 2017 task 1, acoustic scene classification.” <http://dcase.community/challenge2017/task-acoustic-scene-classification-results>, 2017. Accessed: 2021-05-24.
- [69] “Dcase 2018 task 2, general-purpose audio tagging of freesound content with audioset labels.” <http://dcase.community/challenge2018/task-general-purpose-audio-tagging>, 2018. Accessed: 2021-05-24.
- [70] “Dcase 2017 task 4, large-scale weakly supervised sound event detection for smart cars.” <http://dcase.community/challenge2017/task-large-scale-sound-event-detection>, 2017. Accessed: 2021-05-24.
- [71] M. Huzaifah, “Comparison of time-frequency representations for environmental sound classification using convolutional neural networks,” *arXiv preprint arXiv:1706.07156*, 2017.
- [72] Z. Mushtaq, S.-F. Su, and Q.-V. Tran, “Spectral images based environmental sound classification using cnn with meaningful data augmentation,” *Applied Acoustics*, vol. 172, p. 107581, 2021.
- [73] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, “Densely connected convolutional networks,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708, 2017.
- [74] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.
- [75] J. Sharma, O.-C. Granmo, and M. Goodwin, “Environment sound classification using multiple feature channels and deep convolutional neural networks,” *arXiv preprint arXiv:1908.11219*, 2019.
- [76] A. Guzhov, F. Raue, J. Hees, and A. Dengel, “Esresnet: Environmental sound classification based on visual domain models,” in *2020 25th International Conference on Pattern Recognition (ICPR)*, pp. 4933–4940, IEEE, 2021.

- [77] M. Mohaimenuzzaman, C. Bergmeir, I. T. West, and B. Meyer, “Environmental sound classification on the edge: A pipeline for deep acoustic networks on extremely resource-constrained devices,” *arXiv preprint arXiv:2103.03483*, 2021.
- [78] H. Park and C. D. Yoo, “Cnn-based learnable gammatone filterbank and equal-loudness normalization for environmental sound classification,” *IEEE Signal Processing Letters*, vol. 27, pp. 411–415, 2020.
- [79] J. J. Huang and J. J. A. Leanos, “Aclnet: efficient end-to-end audio classification cnn,” *arXiv preprint arXiv:1811.06669*, 2018.
- [80] Z. Mushtaq and S.-F. Su, “Environmental sound classification using a regularized deep convolutional neural network with data augmentation,” *Applied Acoustics*, vol. 167, p. 107389, 2020.
- [81] J. Sharma, O.-C. Granmo, and M. Goodwin, “Environment sound classification using multiple feature channels and attention based deep convolutional neural network,” in *INTERSPEECH*, pp. 1186–1190, 2020.
- [82] Y. Su, K. Zhang, J. Wang, D. Zhou, and K. Madani, “Performance analysis of multiple aggregated acoustic features for environment sound classification,” *Applied Acoustics*, vol. 158, p. 107050, 2020.
- [83] Z. Zhang, S. Xu, S. Cao, and S. Zhang, “Deep convolutional neural network with mixup for environmental sound classification,” in *Chinese conference on pattern recognition and computer vision (prcv)*, pp. 356–367, Springer, 2018.
- [84] J. Leben, “Marsyas music analysis, retrieval and synthesis for audio signals.” <http://marsyas.info/downloads/datasets.html>, 2015. Accessed: 2021-06-02.
- [85] G. Tzanetakis and P. Cook, “Musical genre classification of audio signals,” *IEEE Transactions on speech and audio processing*, vol. 10, no. 5, pp. 293–302, 2002.
- [86] F. Gouyon, S. Dixon, E. Pampalk, and G. Widmer, “Evaluating rhythmic descriptors for musical genre classification,” in *Proceedings of the AES 25th International Conference*, pp. 196–204, 2004.
- [87] E. Law, K. West, M. I. Mandel, M. Bay, and J. S. Downie, “Evaluation of algorithms using games: The case of music tagging,” in *ISMIR*, pp. 387–392, 2009.
- [88] paperswithcode, “Magnatagatune.” <https://paperswithcode.com/dataset/magnatagatune>. Accessed: 2021-06-04.
- [89] Q. Xi, R. M. Bittner, J. Pauwels, X. Ye, and J. P. Bello, “Guitarset: A dataset for guitar transcription,” in *ISMIR*, pp. 453–460, 2018.
- [90] Q. Kong, B. Li, J. Chen, and Y. Wang, “Giantmidi-piano: A large-scale midi dataset for classical piano music,” *arXiv preprint arXiv:2010.07061*, 2020.
- [91] J. Thickstun, Z. Harchaoui, and S. Kakade, “Learning features of music from scratch,” *arXiv preprint arXiv:1611.09827*, 2016.
- [92] C. Liu, L. Feng, G. Liu, H. Wang, and S. Liu, “Bottom-up broadcast neural network for music genre classification,” *Multimedia Tools and Applications*, vol. 80, no. 5, pp. 7313–7331, 2021.
- [93] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going deeper with convolutions,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9, 2015.

- [94] W. W. Ng, W. Zeng, and T. Wang, "Multi-level local feature coding fusion for music genre recognition," *IEEE Access*, vol. 8, pp. 152713–152727, 2020.
- [95] L. Nanni, Y. M. Costa, R. L. Aguiar, C. N. Silla Jr, and S. Brahnam, "Ensemble of deep learning, visual and acoustic features for music genre classification," *Journal of New Music Research*, vol. 47, no. 4, pp. 383–397, 2018.
- [96] D. Ghosal and M. H. Kolekar, "Musical genre and style recognition using deep neural networks and transfer learning," in *Proceedings, APSIPA Annual Summit and Conference*, vol. 2018, pp. 12–15, 2018.
- [97] S. Allamy and A. L. Koerich, "1d cnn architectures for music genre classification," *arXiv preprint arXiv:2105.07302*, 2021.
- [98] R. Yang, L. Feng, H. Wang, J. Yao, and S. Luo, "Parallel recurrent convolutional neural networks-based music genre classification method for mobile devices," *IEEE Access*, vol. 8, pp. 19629–19637, 2020.
- [99] C. Senac, T. Pellegrini, F. Mouret, and J. Pinquier, "Music feature maps with convolutional neural networks for music genre classification," in *Proceedings of the 15th International Workshop on Content-Based Multimedia Indexing*, pp. 1–5, 2017.
- [100] H. Tang and N. Chen, "Combining cnn and broad learning for music classification," *IEICE Transactions on Information and Systems*, vol. 103, no. 3, pp. 695–701, 2020.
- [101] Y. Yu, S. Luo, S. Liu, H. Qiao, Y. Liu, and L. Feng, "Deep attention based music genre classification," *Neurocomputing*, vol. 372, pp. 84–91, 2020.
- [102] S. R. Livingstone and F. A. Russo, "The ryerson audio-visual database of emotional speech and song (ravdess): A dynamic, multimodal set of facial and vocal expressions in north american english," *PloS one*, vol. 13, no. 5, p. e0196391, 2018.
- [103] K. Becker, "Gender recognition by voice identify a voice as male or female." <https://www.kaggle.com/primaryobjects/voicegender>, 2016. Accessed: 2021-05-31.
- [104] A. Nagrani, J. S. Chung, and A. Zisserman, "Voxceleb: a large-scale speaker identification dataset," *arXiv preprint arXiv:1706.08612*, 2017.
- [105] P. Warden, "Launching the speech commands dataset." <https://ai.googleblog.com/2017/08/launching-speech-commands-dataset.html>, Aug. 2017. Accessed: 2021-05-31.
- [106] P. Warden, "Speech commands: A dataset for limited-vocabulary speech recognition," *arXiv preprint arXiv:1804.03209*, 2018.
- [107] K. Park and T. Mulc, "Css10: A collection of single speaker speech datasets for 10 languages," *arXiv preprint arXiv:1903.11269*, 2019.
- [108] B. Tura, F. Diego, C. Segura, and J. Luque, "Efficient keyword spotting through long-range interactions with temporal lambda networks," *arXiv preprint arXiv:2104.08086*, 2021.
- [109] S. Majumdar and B. Ginsburg, "Matchboxnet-1d time-channel separable convolutional neural network architecture for speech commands recognition," *arXiv preprint arXiv:2004.08531*, 2020.
- [110] Y. Wei, Z. Gong, S. Yang, K. Ye, and Y. Wen, "Edgecnn: an edge-computing oriented model of acoustic feature enhancement for keyword spotting," *Journal of Ambient Intelligence and Humanized Computing*, pp. 1–11, 2021.

- [111] S. Mittermaier, L. Kürzinger, B. Waschneck, and G. Rigoll, “Small-footprint keyword spotting on raw audio data with sinc-convolutions,” in *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 7454–7458, IEEE, 2020.
- [112] D. C. de Andrade, S. Leo, M. L. D. S. Viana, and C. Bernkopf, “A neural attention model for speech command recognition,” *arXiv preprint arXiv:1808.08929*, 2018.
- [113] D. Peter, W. Roth, and F. Pernkopf, “End-to-end keyword spotting using neural architecture search and quantization,” *arXiv preprint arXiv:2104.06666*, 2021.
- [114] S. Choi, S. Seo, B. Shin, H. Byun, M. Kersner, B. Kim, D. Kim, and S. Ha, “Temporal convolution for real-time keyword spotting on mobile devices,” *arXiv preprint arXiv:1904.03814*, 2019.
- [115] A. Soliman, S. Mohamed, and I. A. Abdelrahman, “Isolated word speech recognition using convolutional neural network,” in *2020 International Conference on Computer, Control, Electrical, and Electronics Engineering (ICCCEEE)*, pp. 1–6, IEEE, 2021.
- [116] R. Tang and J. Lin, “Deep residual learning for small-footprint keyword spotting,” in *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 5484–5488, IEEE, 2018.
- [117] A. Berg, M. O’Connor, and M. T. Cruz, “Keyword transformer: A self-attention model for keyword spotting,” *arXiv preprint arXiv:2104.00769*, 2021.
- [118] “Birdclef 2021 - main task.” <https://www.imageclef.org/BirdCLEF2021MainTask>, Jan. 2021. Accessed: 2021-02-12.
- [119] D. Stowell, M. Wood, Y. Stylianou, and H. Glotin, “Bird detection in audio: a survey and a challenge,” in *2016 IEEE 26th International Workshop on Machine Learning for Signal Processing (MLSP)*, pp. 1–6, IEEE, 2016.
- [120] V. Lostanlen, J. Salamon, A. Farnsworth, S. Kelling, and J. P. Bello, “Birdvox-full-night: A dataset and benchmark for avian flight call detection,” in *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 266–270, IEEE, 2018.
- [121] D. Stowell and M. D. Plumbley, “An open dataset for research on audio field recording archives: freefield1010,” *arXiv preprint arXiv:1309.5275*, 2013.
- [122] X. canto Foundation, “xeno-canto : Sharing bird sounds from around the world.” <https://www.xeno-canto.org/>, 2005. Accessed: 2020-09-12.
- [123] “Birdclef 2020.” <https://www.imageclef.org/BirdCLEF2021MainTask>, 2020. Accessed: 2020-08-27.
- [124] S. Kahl, F.-R. Stöter, H. Goëau, H. Glotin, R. Planque, W.-P. Vellinga, and A. Joly, “Overview of birdclef 2019: large-scale bird recognition in soundscapes,” in *Working Notes of CLEF 2019-Conference and Labs of the Evaluation Forum*, pp. 1–9, CEUR, 2019.
- [125] S. Kahl, M. Clapp, W. Hopping, H. Goëau, H. Glotin, R. Planqué, W.-P. Vellinga, and A. Joly, “Overview of birdclef 2020: Bird sound recognition in complex acoustic environments,” in *CLEF 2020*, 2020.
- [126] “Dcase2018 challenge iee aasp challenge on detection and classification of acoustic scenes and events.” <http://dcase.community/challenge2018/index>, Mar. 2018. Accessed: 2021-04-22.

- [127] M. Lasseck, “Acoustic bird detection with deep convolutional neural networks,” in *Proceedings of the Detection and Classification of Acoustic Scenes and Events 2018 Workshop (DCASE2018)*, pp. 143–147, 2018.
- [128] M. Lasseck, “Audio-based bird species identification with deep convolutional neural networks,” in *CLEF (Working Notes)*, 2018.
- [129] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, “Rethinking the inception architecture for computer vision,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2818–2826, 2016.
- [130] M. Lasseck, “Bird species identification in soundscapes,” in *CLEF (Working Notes)*, 2019.
- [131] M. Mühling, J. Franz, N. Korfhage, and B. Freisleben, “Bird species recognition via neural architecture search,” *CLEF Working Notes*, 2020.
- [132] “Rainforest connection | protecting rainforests with real-time data.” <https://www.rfcx.org>, 2021. Accessed: 2021-02-15.
- [133] Amah, “Rainforest connection : lutter contre la déforestation avec des smartphones recyclés.” <https://www.geekmaispasque.com/2015/09/rainforest-connection-lutter-contre-la-deforestation-avec-des-smartphones-recycles/>, Sep. 2015. Accessed: 2021-02-17.
- [134] Amah, “Borror laboratory of bioacoustics.” <https://blb.osu.edu/>, 2021. Accessed: 2021-02-12.
- [135] N. Douglas, “The borror laboratory of bioacoustics: a non-traditional museum collection.” <https://u.osu.edu/biomuseum/2015/09/25/the-borror-laboratory-of-bioacoustics-a-non-traditional-museum-collection/>, 2015. Accessed: 2021-02-21.
- [136] Mikołajczyk, A., “xcdl.py [github repository].” <https://github.com/AgaMiko/xeno-canto-download/blob/master/xcdl.py>, 2019.
- [137] “Gunshot audio forensics dataset – gunshot audio analysis.” <https://cadreforensics.com/audio/>. Accessed: 2020-11-24.
- [138] G. S.-V. . M. W. Marcelo Araya-Salas, “Cornell lab or ornithology - why wave (.wav) files?.” <https://www.macauleylibrary.org/resources/why-wav/>, 2017. Accessed: 2020-12-12.
- [139] M. Lasseck, “Bird song classification in field recordings: winning solution for nips4b 2013 competition,” in *Proc. of int. symp. Neural Information Scaled for Bioacoustics, sabiod. org/nips4b, joint to NIPS, Nevada*, pp. 176–181, 2013.
- [140] E. Sprengel, M. Jaggi, Y. Kilcher, and T. Hofmann, “Audio based bird species identification using deep learning techniques,” tech. rep., Eidgenössische Technische Hochschule (ETH) Zürich, 2016.
- [141] kang & atul, “Tag archives: Cv2.medianblur().” <https://theailearner.com/tag/cv2-medianblur/>, May 2019. Accessed: 2021-02-12.
- [142] Hsuan-Gu Chou, J, “aud_to_spec.py [github repository].” https://github.com/Mipanox/Bird_cocktail/blob/master/codes/aud_to_spec.py, 2018.
- [143] Geena, K., “which_animal [github repository].” https://github.com/libphy/which_animal, 2016.

- [144] S. Kahl, T. Wilhelm-Stein, H. Hussein, H. Klinck, D. Kowerko, M. Ritter, and M. Eibl, "Large-scale bird sound classification using convolutional neural networks.," in *CLEF (Working Notes)*, 2017.
- [145] D. Stowell, M. D. Wood, H. Pamuła, Y. Stylianou, and H. Glotin, "Automatic acoustic detection of birds through deep learning: the first bird audio detection challenge," *Methods in Ecology and Evolution*, vol. 10, no. 3, pp. 368–380, 2019.
- [146] A. Madhavi and R. Pamnani, "Deep learning based audio classifier for bird species," *International Journal of Scientific Development and Research (IJS DR)*, vol. 3, no. 10, pp. 228–233, 2018.
- [147] Keras, "Learning rate schedules api." https://keras.io/api/optimizers/learning_rate_schedules/. Accessed: 2020-09-20.
- [148] Keras, "Earlystopping." https://keras.io/api/callbacks/early_stopping/. Accessed: 2020-09-20.
- [149] Keras, "Callbacks api." <https://keras.io/api/callbacks/>. Accessed: 2020-09-20.
- [150] K. Palanisamy, D. Singhania, and A. Yao, "Rethinking cnn models for audio classification," *arXiv preprint arXiv:2007.11154*, 2020.
- [151] A. I. Mohammed and A. A. Tahir, "A new optimizer for image classification using wide resnet (wrn)," *Academic Journal of Nawroz University*, vol. 9, no. 4, pp. 1–13, 2020.
- [152] D. Soydaner, "A comparison of optimization algorithms for deep learning," *arXiv preprint arXiv:2007.14166*, 2020.
- [153] K. J. Piczak, "Recognizing bird species in audio recordings using deep convolutional neural networks.," in *CLEF (Working Notes)*, pp. 534–543, 2016.
- [154] Clark, M., "keras_weighted_categorical_crossentropy.py [github repository]." <https://gist.github.com/wassname/ce364fddfc8a025bfab4348cf5de852d>, 2017.
- [155] R. L. Galvez, A. A. Bandala, E. P. Dadios, R. R. P. Vicerra, and J. M. Z. Maningo, "Object detection using convolutional neural networks," in *TENCON 2018-2018 IEEE Region 10 Conference*, pp. 2023–2027, IEEE, 2018.
- [156] P. N. Whatmough, C. Zhou, P. Hansen, S. K. Venkataramanaiah, J.-s. Seo, and M. Mattina, "Fixynn: Efficient hardware for mobile computer vision via transfer learning," *arXiv preprint arXiv:1902.11128*, 2019.
- [157] X. Liu, Z. Jia, X. Hou, M. Fu, L. Ma, and Q. Sun, "Real-time marine animal images classification by embedded system based on mobilenet and transfer learning," in *OCEANS 2019-Marseille*, pp. 1–5, IEEE, 2019.
- [158] J. Dong, K. Ota, and M. Dong, "Real-time survivor detection in uav thermal imagery based on deep learning," in *2020 16th International Conference on Mobility, Sensing and Networking (MSN)*, pp. 352–359, IEEE, 2020.
- [159] N. Van Hieu and N. L. H. Hien, "Automatic plant image identification of vietnamese species using deep learning models," *arXiv preprint arXiv:2005.02832*, 2020.
- [160] N. Instruments, "Software filtering: Windowing - general analog concepts." <https://www.ni.com/fr-be/support/documentation/supplemental/06/software-filtering--windowing---general-analog-concepts.html>, June 2021. Accessed: 2020-09-12.

- [161] M. Allen, “Exploring the best possible trade-off between competing objectives: identifying the pareto front.” <https://pythonhealthcare.org/2018/09/27/93-exploring-the-best-possible-trade-off-between-competing-objectives-identifying-the> Sept. 2018. Accessed: 2021-05-27.
- [162] L. Nanni, G. Maguolo, and M. Paci, “Data augmentation approaches for improving animal audio classification,” *Ecological Informatics*, vol. 57, p. 101084, 2020.
- [163] Jordal, I., “audiomentations [github repository].” <https://github.com/iver56/audiomentations>, 2021.
- [164] Malek, A., “pydiogment [github repository].” <https://github.com/SuperKogito/pydiogment>, 2020.
- [165] J. Schlüter and T. Grill, “Exploring data augmentation for improved singing voice detection with neural networks,” in *ISMIR*, pp. 121–126, 2015.
- [166] S. Bhardwaj, *Audio Data Augmentation with respect to Musical Instrument Recognition*. PhD thesis, MA thesis. 2017. DOI: <https://doi.org/10.5281/zenodo.1066137>, 2017.
- [167] S. Y. Mudugandla, “10 hyperparameter optimization frameworks.” <https://pythonhealthcare.org/2018/09/27/93-exploring-the-best-possible-trade-off-between-competing-objectives> Sept. 2020. Accessed: 2021-04-04.
- [168] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica, “Tune: A research platform for distributed model selection and training,” *arXiv preprint arXiv:1807.05118*, 2018.
- [169] Autonomio, “talos [github repository].” <https://github.com/autonomio/talos>, 2020.
- [170] Autonomio, “optuna [github repository].” <https://github.com/optuna/optuna>, 2021.
- [171] P. Gabrys, “How to make your model awesome with optuna.” <https://towardsdatascience.com/how-to-make-your-model-awesome-with-optuna-b56d490368af>, Feb. 2019. Accessed: 2021-04-09.
- [172] G. Lu, “Tuning hyperparameters with optuna.” <https://towardsdatascience.com/tuning-hyperparameters-with-optuna-af342facc549>, Mar. 2021. Accessed: 2021-04-09.
- [173] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, “Optuna: A next-generation hyperparameter optimization framework,” in *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 2623–2631, 2019.
- [174] A. Bissuel, “Hyper-parameter optimization algorithms: a short review.” <https://medium.com/criteo-engineering/hyper-parameter-optimization-algorithms-2fe447525903>, Apr. 2019. Accessed: 2021-04-08.
- [175] J. Brownlee, “Understand the impact of learning rate on neural network performance.” <https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/>, Jan. 2019. Accessed: 2020-11-09.
- [176] M. Braquet, D. Bol, and R. Sadre, *Design of an ultra-low-power energy-harvesting audio sensor for ecosystem monitoring*. PhD thesis, Master Thesis. UCL-Ecole polytechnique de Louvain, 2020.
- [177] Arm, “Intrinsic functions for simd instructions [only cortex-m4 and cortex-m7].” https://www.keil.com/pack/doc/CMSIS/Core/html/group__intrinsic__SIMD__gr.html#details, 2019. Accessed: 2021-03-11.

- [178] STM32 Core Team, *UM2526: Getting started with x-cube-ai expansion package for artificial intelligence (ai)*. STMicroelectronics, 2019.
- [179] ARM, “Mdk microcontroller development kit.” <https://www2.keil.com/mdk5>. Accessed: 2021-03-02.
- [180] ARM, “Architecting a smarter world.” <https://www.arm.com/>. Accessed: 2021-02-24.
- [181] C. Zhang, “Cifar-10 image classification.” <https://becominghuman.ai/cifar-10-image-classification-fd2ace47c5e8>, Aug. 2020. Accessed: 2021-05-31.
- [182] C. Zhang, “How to run deep learning model on microcontroller with cmsis-nn (part 2).” <https://www.dlology.com/blog/how-to-run-deep-learning-model-on-microcontroller-with-cmsis-nn-part-2/>, Sept. 2018. Accessed: 2021-03-03.
- [183] L. A. dos Santos, “Makingfaster.” https://leonardoaraujosantos.gitbook.io/artificial-inteligence/machine_learning/deep_learning/convolution_layer/making_faster, 2020. Accessed: 2021-04-23.
- [184] Y. Jain, “Speeding up convolutions.” <https://scocoyash.github.io/speeding-up-convolutions/#example-for-im2col>, May 2020. Accessed: 2021-05-12.
- [185] Arm Ltd., “Cmsis nn software library.” https://arm-software.github.io/CMSIS_5/NN/html/group__NNConv.html, 2020. Accessed: 2021-03-03.
- [186] D. Lin, S. Talathi, and S. Annapureddy, “Fixed point quantization of deep convolutional networks,” in *International conference on machine learning*, pp. 2849–2858, PMLR, 2016.
- [187] L. Lai, N. Suda, and V. Chandra, “Deep convolutional neural network inference with floating-point weights and fixed-point activations,” *arXiv preprint arXiv:1703.03073*, 2017.
- [188] Wikipedia, “Q (number format).” <https://scocoyash.github.io/speeding-up-convolutions/#example-for-im2col>, Mar. 2021. Accessed: 2021-03-12.
- [189] A. Capotondi, M. Rusci, M. Fariselli, and L. Benini, “Cmix-nn: Mixed low-precision cnn library for memory-constrained edge devices,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 5, pp. 871–875, 2020.
- [190] A. Capotondi, M. Rusci, M. Fariselli, and L. Benini, “Cmix-nn [github repository].” <https://github.com/EEESlab/CMix-NN>, 2020.
- [191] A. Garofalo, M. Rusci, F. Conti, D. Rossi, and L. Benini, “Pulp-nn: accelerating quantized neural networks on parallel ultra-low-power risc-v processors,” *Philosophical Transactions of the Royal Society A*, vol. 378, no. 2164, p. 20190155, 2020.
- [192] A. Garofalo, M. Rusci, F. Conti, D. Rossi, and L. Benini, “pulp-nn [github repository].” <https://github.com/pulp-platform/pulp-nn>, 2021.
- [193] Arm Software, “Cmsis-nn cifar10 example [github repository].” <https://github.com/ARM-software/ML-examples/tree/master/cmsisnn-cifar10>, 2019.
- [194] Quinn (quinnabrvau), “Keras_2_cmsis [github repository].” https://github.com/quinnabrvau/Keras_2_CMSIS, 2019.
- [195] Arm Ltd., “Converting a neural network for arm cortex-m with cmsis-nn.” <https://developer.arm.com/solutions/machine-learning-on-arm/developer-material/how-to-guides/converting-a-neural-network-for-arm-cortex-m-with-cmsis-nn/single-page>. Accessed: 2021-03-03.

- [196] B. Moons, “Quantizedneuralnetworks-keras-tensorflow [github repository].” <https://github.com/BertMoons/QuantizedNeuralNetworks-Keras-Tensorflow>, 2017.
- [197] R. Krishnamoorthi, “Quantizing deep convolutional networks for efficient inference: A whitepaper,” *arXiv preprint arXiv:1806.08342*, 2018.
- [198] C. A. G. Pierre, “Batch norm folding: An easy way to improve your network speed.” <https://scortex.io/batch-norm-folding-an-easy-way-to-improve-your-network-speed/>, June 2020. Accessed: 2021-03-27.
- [199] D. Mishkin, “caffenet-benchmar [github repository].” <https://github.com/ducha-aiki/caffenet-benchmark/blob/master/batchnorm.md>, 2016.
- [200] Arm Software, “Cmsis_5 [github repository].” https://github.com/ARM-software/CMSIS_5/blob/develop/CMSIS/NN/Scripts/NNFunctions/fully_connected_opt_weight_generation.py, 2017.
- [201] S. Migacz, “8-bit inference with tensorrt.” <https://on-demand.gputechconf.com/gtc/2017/presentation/s7310-8-bit-inference-with-tensorrt.pdf>, May 2018. Accessed: 2021-03-30.
- [202] R. Banner, Y. Nahshan, E. Hoffer, and D. Soudry, “Post-training 4-bit quantization of convolution networks for rapid-deployment,” *arXiv preprint arXiv:1810.05723*, 2018.
- [203] Arm Software, “ML-kws-for-mcu [github repository].” https://github.com/ARM-software/ML-KWS-for-MCU/blob/master/Deployment/Quant_guide.md, 2018.
- [204] Arm Software, “ML-examples [github repository].” <https://github.com/ARM-software/ML-examples/blob/master/cmsisnn-cifar10/code/m4/nn.cpp>, 2018.
- [205] ARM-software, “cifar10 [github repository].” https://github.com/ARM-software/CMSIS_5/blob/develop/CMSIS/NN/Examples/ARM/arm_nn_examples/cifar10/arm_nnexamples_cifar10.cpp, 2019.
- [206] J. Labrosse, “Measuring code execution time on arm cortex-m mcus.” <https://www.embeddedcomputing.com/technology/processing/measuring-code-execution-time-on-arm-cortex-m-mcus>, July 2018. Accessed: 2021-05-01.
- [207] S. P. Singh, “Convolution layer: The layer that takes over 70% of time in a machine learning model.” <https://iq.opengenus.org/convolution-layer/>, 2021. Accessed: 2021-05-30.
- [208] S. Gharib, D. Bol, and M. Lefebvre, *Polyp classification for detection of colorectal cancer with capsule endoscopy*. PhD thesis, Master Thesis. UCL-Ecole polytechnique de Louvain, 2020.
- [209] Ambiq Micro, *Apollo3 Blue MCU Datasheet*, 2021. Revision 0.15.0.
- [210] Jessie, “Comparison between different memories(pcm, stt, ram, sram, dram, flash nand, hdd).” <https://www.elinfor.com/knowledge/comparison-between-different-memoriespcm-stt-ram-sram-dram-flash-nand-hdd-p-10908>, Jan. 2019. Accessed: 2020-05-20.
- [211] S. Colley, “Real time clocks (rtcs) in microcontroller timers.” <https://www.allaboutcircuits.com/technical-articles/introduction-to-microcontroller-timers-real-time-clocks/>, Feb. 2020. Accessed: 2021-05-30.
- [212] tektronix, *2450 SourceMeter ® SMU Instrument Datasheet*.
- [213] gwinstek, *GDS-1000U-Series 100MHz/70MHz/50MHz Digital Storage Oscilloscope*.

- [214] S. Engineering, “Clock gating dynamic power reduction by gating the clock.” https://semiengineering.com/knowledge_centers/low-power/techniques/clock-gating-2/. Accessed: 2021-05-30.
- [215] D. Bol, “Low-power basics & embedded memories.” Slides, 2021.
- [216] T. I. Team, “Cmos power consumption and cpd calculation,” Tech. Rep. SCAA035B, Texas Instruments, June 1997.
- [217] A. Wattiez, *Design of an ultra-low power always-on analog front-end for audio events detection in forests*. PhD thesis, Master Thesis. UCL-Ecole polytechnique de Louvain, 2021.
- [218] Wikipedia, “Low-noise amplifier.” https://en.wikipedia.org/wiki/Low-noise_amplifier, Jan. 2021. Accessed: 2021-06-03.
- [219] T. Hess, *Ultra-Low-Power Over-the-Air-Update in secure LoRaWAN networks*. PhD thesis, Master Thesis. UCL-Ecole polytechnique de Louvain, 2020.
- [220] W. Shan, M. Yang, T. Wang, Y. Lu, H. Cai, L. Zhu, J. Xu, C. Wu, L. Shi, and J. Yang, “A 510-nw wake-up keyword-spotting chip using serial-fft-based mfcc and binarized depthwise separable cnn in 28-nm cmos,” *IEEE Journal of Solid-State Circuits*, vol. 56, no. 1, pp. 151–164, 2020.
- [221] D. Wang, S. J. Kim, M. Yang, A. A. Lazar, and M. Seok, “A background-noise and process-variation-tolerant 109nw acoustic feature extractor based on spike-domain divisive-energy normalization for an always-on keyword spotting device,” in *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 64, pp. 160–162, IEEE, 2021.
- [222] S. Jeong, Y. Chen, T. Jang, J. Tsai, D. Blaauw, H.-S. Kim, and D. Sylvester, “A 12nw always-on acoustic sensing and object recognition microsystem using frequency-domain feature extraction and svm classification,” in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 362–363, IEEE, 2017.
- [223] Q. Li, Y. Yang, T. Lan, H. Zhu, Q. Wei, F. Qiao, X. Liu, and H. Yang, “Msp-mfcc: Energy-efficient mfcc feature extraction method with mixed-signal processing architecture for wearable speech recognition applications,” *IEEE Access*, vol. 8, pp. 48720–48730, 2020.
- [224] Y. Sudo, K. Itoyama, K. Nishida, and K. Nakadai, “Environmental sound segmentation utilizing mask u-net,” in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 5340–5345, IEEE, 2019.
- [225] Y. Sudo, K. Itoyama, K. Nishida, and K. Nakadai, “Multi-channel environmental sound segmentation utilizing sound source localization and separation u-net,” in *2021 IEEE/SICE International Symposium on System Integration (SII)*, pp. 382–387, IEEE, 2021.
- [226] M. D. Zeiler and R. Fergus, “Visualizing and understanding convolutional networks,” in *European conference on computer vision*, pp. 818–833, Springer, 2014.
- [227] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, “Grad-cam: Visual explanations from deep networks via gradient-based localization,” in *Proceedings of the IEEE international conference on computer vision*, pp. 618–626, 2017.
- [228] D. Hendrycks and K. Gimpel, “A baseline for detecting misclassified and out-of-distribution examples in neural networks,” *arXiv preprint arXiv:1610.02136*, 2016.
- [229] S. Liang, Y. Li, and R. Srikant, “Enhancing the reliability of out-of-distribution image detection in neural networks,” *arXiv preprint arXiv:1706.02690*, 2017.

- [230] T. DeVries and G. W. Taylor, “Learning confidence for out-of-distribution detection in neural networks,” *arXiv preprint arXiv:1802.04865*, 2018.
- [231] G. Nicolae, A. Gaiță, A. Rădoi, and C. Burileanu, “A method for chainsaw sound detection based on haar-like features,” in *2018 41st International Conference on Telecommunications and Signal Processing (TSP)*, pp. 1–5, IEEE, 2018.
- [232] J. D. C. Arevalo, P. C. Calica, B. A. D. Celestino, K. A. Dimapunong, D. J. D. Lopez, and Y. D. Austria, “Towards real-time illegal logging monitoring: Gas-powered chainsaw logging detection system using k-nearest neighbors,” in *2020 IEEE 10th International Conference on System Engineering and Technology (ICSET)*, pp. 156–160, IEEE, 2020.
- [233] V. Singh, K. C. Ray, and S. Tripathy, “Robust gunshot features and its classification using support vector machine for wildlife protection,” in *Electronic Systems and Intelligent Computing*, pp. 939–948, Springer, 2020.
- [234] S. U. Rahman, A. Khan, S. Abbas, F. Alam, and N. Rashid, “Hybrid system for automatic detection of gunshots in indoor environment,” *Multimedia Tools and Applications*, vol. 80, no. 3, pp. 4143–4153, 2021.
- [235] J. Bajzik, J. Prinosil, and D. Koniar, “Gunshot detection using convolutional neural networks,” in *2020 24th International Conference Electronics*, pp. 1–5, IEEE, 2020.
- [236] F. Naccari, I. Guarneri, S. Curti, and A. A. Savi, “Embedded acoustic scene classification for low power microcontroller devices,”
- [237] M. Meyer, T. Farei-Campagna, A. Pasztor, R. Da Forno, T. Gsell, J. Faillettaz, A. Vieli, S. Weber, J. Beutel, and L. Thiele, “Event-triggered natural hazard monitoring with convolutional neural networks on the edge,” in *2019 18th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pp. 73–84, IEEE, 2019.
- [238] M. Eggimann, J. Erb, P. Mayer, M. Magno, and L. Benini, “Low power embedded gesture recognition using novel short-range radar sensors,” in *2019 IEEE SENSORS*, pp. 1–4, IEEE, 2019.
- [239] G. Cerutti, R. Prasad, and E. Farella, “Convolutional neural network on embedded platform for people presence detection in low resolution thermal images,” in *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 7610–7614, IEEE, 2019.
- [240] M. Kortas, “Sound-based bird classification.” <https://towardsdatascience.com/sound-based-bird-classification-965d0ecacb2b>, Jan. 2020. Accessed: 2021-02-12.
- [241] J. Hsuan-Gu Chou, “Cocktail party problem of bird sounds [github repository].” https://github.com/MipanoX/Bird_cocktail, Mar. 2018.
- [242] T. Kokkinidis, “Wildfire in greece destroys over 4,000 acres of forest.” <https://greekreporter.com/2021/05/21/wildfire-in-greece-destroys-over-4000-acres-of-forest/>, May 2021. Accessed: 2021-04-12.
- [243] DCNEPAL, “Wildfires destroy 1200 hectares of 47 community forests in myagdi.” <https://english.dcnepal.com/2021/04/07/wildfires-destroy-1200-hectares-forest-at-47-community-forests-in-myagdi/>, Apr. 2021. Accessed: 2021-04-12.
- [244] J. Van Bruggen & al., “Wildfires destroy more than forests.” <http://glossyinc.com/2018/09/10/wildfires-destroy-forests/>, Oct. 2018. Accessed: 2021-04-12.

- [245] S. Shah, and A. Shah, “Fighting fire with machine learning: two students use tensorflow to predict wildfires.” <https://www.blog.google/technology/ai/fighting-fire-machine-learning-two-students-use-tensorflow-predict-wildfires/>, June 2018. Accessed: 2021-04-12.
- [246] S. Lau, “Learning rate schedules and adaptive learning rate methods for deep learning.” <https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f433>, July 2017. Accessed: 2020-02-12.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl