Norwegian University
of Life Sciences

Master's Thesis 2019 30 ECTS
Faculty of Science and Technology

Environmental Sound Classification on Microcontrollers using Convolutional Neural Networks

Jon Nordby
Master of Science in Data Science

Abstract

Noise is a growing problem in urban areas, and according to the WHO is the second environmental cause of health problems in Europe. Noise monitoring using Wireless Sensor Networks are being applied in order to understand and help mitigate these noise problems. It is desirable that these sensor systems, in addition to logging the sound level, can indicate what the likely sound source is. However, transmitting audio to a cloud system for classification is energy-intensive and may cause privacy issues. It is also critical for widespread adoption and dense sensor coverage that individual sensor nodes are low-cost. Therefore we propose to perform the noise classification on the sensor node, using a low-cost microcontroller.

Several Convolutional Neural Networks were designed for the STM32L476 low-power microcontroller using the Keras deep-learning framework, and deployed using the vendor-provided X-CUBE-AI inference engine. The resource budget for the model was set at maximum 50% utilization of CPU, RAM, and FLASH. 10 model variations were evaluated on the Environmental Sound Classification task using the standard Urbansound8k dataset.

The best models used Depthwise-Separable convolutions with striding for downsampling, and were able to reach 70.9% mean 10-fold accuracy while consuming only 20% CPU. To our knowledge, this is the highest reported performance on Urbansound8k using a microcontroller. One of the models was also tested on a microcontroller development device, demonstrating the classification of environmental sounds in real-time.

These results indicate that it is computationally feasible to classify environmental sound on low-power microcontrollers. Further development should make it possible to create wireless sensor-networks for noise monitoring with on-edge noise source classification.

Contents

List of Tables

List of Figures

1 | Introduction

1.1 Environmental noise

Noise is a growing problem in urban areas, and due to increasing urbanization more and more people are affected. Major sources of noise include transportation, construction, industry and recreational activities. The sum of all the noise is referred to as environmental noise or noise pollution.

Noise pollution over sustained periods of time affects health and well-being in many ways. Noise can be a source of annoyance and increased stress, cause sleeping disturbance and increase risk of heart diseases. WHO has estimated that in Europe 1.6 million healthy life years (Disability-Adjusted Life Years, DALY) are lost annually due to noise pollution[1]. This makes noise pollution the second environmental cause of health problems in Europe, after air pollution.

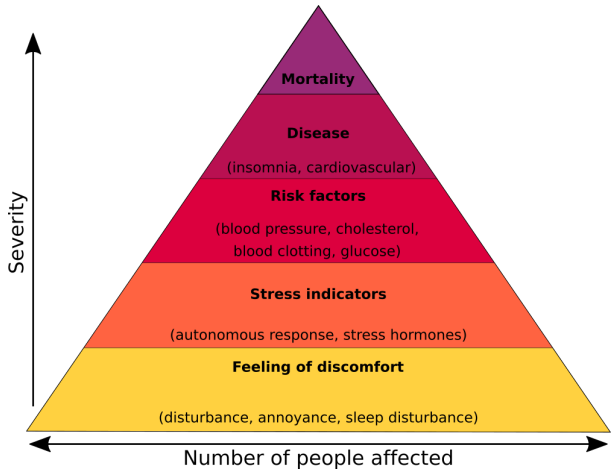


Figure 1.1: Health impacts of noise at different severity levels [2]

In the EU environmental noise is regulated by Environmental Noise Directive (2002/49/EC)[3]. The purpose of the directive is to:

- Determine people's exposure to environmental noise
- Ensuring that information on environmental noise is available to the public
- Preventing and reducing environmental noise where necessary
- Preserving environmental noise quality where it is good

Member States of the EU are required to create noise maps and noise management action plans every 5 years. These must cover all urban areas, major roads, railways and airports over a certain size.

The noise maps are created using simulation of known noise sources (such as car traffic) with mathematical sound propagation models, based on estimates for traffic numbers. These maps only give yearly average noise levels for the day, evening and night.

1.2 Noise Monitoring with Wireless Sensor Networks

Several cities have started to deploy networks of sound sensors in order to better understand and reduce noise issues. These sensor networks consist of many sensor nodes positioned in the area of interest, transmitting the data to a central system for storage and reporting.

Examples of established projects are Dublin City Noise[4] with 14 sensors across the city since 2016. The Sounds of New York City (SONYC)[5] project had 56 sound sensors installed as of 2018[6], and the Barcelona Noise Monitoring System[7] had 86 sound sensors[8]. The CENSE[9] project plans to install around 150 sensors in Lorient, France[10].

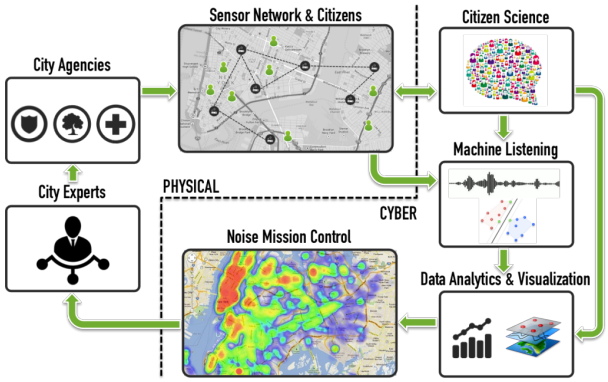


Figure 1.2: Illustration of how Sounds of New York City[11] system combines sensor networks and citizen reporting with data analysis and to present city experts and agencies with a visual interactive dashboard "Noise Mission Control".

To keep costs low and support a dense coverage, the sensor nodes can be designed to operate wirelessly. Communication is done using wireless radio technologies such as WiFi, GSM, NB-IoT or 6LoWPAN. The sensor harvests its energy, often using solar power or from streetlights powered at night. A battery backup allows the sensor to continue operating also when energy is momentarily unavailable.

These sensor networks enable continuous logging of the sound pressure level, measured in Decibel (dB SPL) over a reference pressure level (typically 20×10^{-6}Pa). Since the sound pressure level is continuously varying, it is summarized over a specified time-period using Equivalent Continuous Sound Level (L_{eq}). Typical measurement resolutions are per minute, per second or per 125 ms. Measurements often use A-weighting to approximate the sensitivity of the human ear at different frequencies. In Europe, sound level sensors

are designed to specifications of IEC 61672-1 Sound Level Meters[12], and the standard for North America is ANSI S1.4[13].

Sensors can also provide information that can be used to characterize the noise, for instance to identify the likely noise sources. This is desirable in order to understand the cause of the noise, identify which regulations the noise falls under, which actors may be responsible, and to initiate possible interventions.

This requires much more data than sound level measurements, making it challenging to transmit the data within the bandwidth and energy budget of a wireless sensor. The sensor may also capture sensitive information and violate privacy requirements by recording and storing such detailed data.

To address these concerns several methods for efficiently coding the information before transmitting to the server have been developed. Figure 1.3 shows an overview of the different approaches.

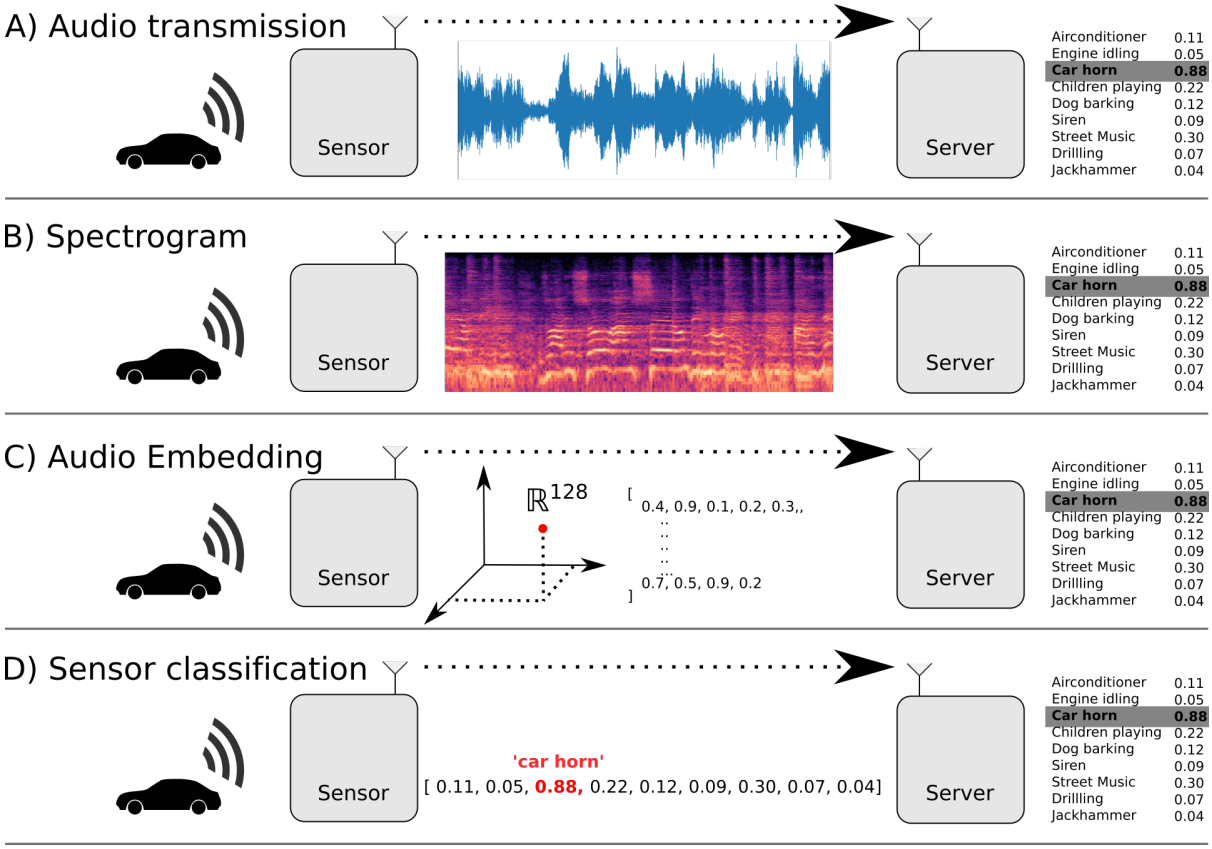


Figure 1.3: Different data transmission strategies for a noise sensor network with noise source classification capability. A) Sensor sends raw audio data with classification on server. B) Sensor sends spectrograms as a intermediate audio representation. Classification on server. C) Sensor sends neural network audio embeddings as intermediate audio representation. Classification on server. D) Sensor performs classification on device and sends result to server. No audio or intermediate representation needs to be transmitted.

In [14], the authors propose a compressed noise profile based on lossy compression of spectrograms. For 125ms time resolution, the bit-rate is between 400 and 1400 bits per

second, however this gave a 5 percentage points reduction in classification accuracy. This is shown as case B) of Figure 1.3.

Others have proposed to use neural networks to produce an audio "embedding" inspired by the success of word embeddings[15] for Natural Language Processing. This is shown as case C) of Figure 1.3. In VGGish[16] model trained on Audioset[17] an 8-bit, 128-dimensional embedding per 1 second, leading to a data rate of 1024 bits per second. L^3 (Look, Listen, Learn)[18] similarly proposed an embedding with 512 dimensions. The computation of such an embedding generally requires very large models and lots of computational resources. $EdgeL^3$[19] showed that the L^3 model can be compressed by up to 95%, however the authors state that more work is needed to fit the RAM constraints of desirable sensor hardware.

The minimal amount of data transmissions would be achieved if the detected noise category was sent, requiring to perform the entire classification on the sensor. This is shown as case D) of Figure 1.3. Such an approach could also eliminate the need to send personally identifiable data to a centralized server.

This motivates the problem statement of this thesis:

> Can we classify environmental sounds directly on a wireless and battery-operated noise sensor?

2 | Background

2.1 Machine Learning

Machine Learning is the use of algorithms and statistical models to effectively perform a task, without having to explicitly program the instructions for how to perform this task. Instead, the algorithms learn to perform the desired function from provided data.

Supervised learning uses a training dataset where each sample is labeled with the correct output. These labels are normally provided by manual annotation by humans inspecting the data, a time-intensive and costly process. In *unsupervised learning*, models are trained without access to labeled data. This is often used for cluster analysis (automatic discovery of sample groups).

Supervised learning techniques can be used for regression and for classification. In regression where the goal is to predict a continuous real-valued variable, and for classification a discrete variable.

2.1.1 Classification

Classification is a machine learning task where the goal is to train a model that can accurately predict which class(es) the data belongs to. Examples use-cases could be to determine from an image which breed a dog is, to predict from a text whether it is positive or negative towards the subject matter - or to determine from audio what kind of sound is present.

In single-label classification, a sample can only belong to a single class. In closed-set classification, the possible class is one of N predetermined classes. Many classification problems are treated as single-label and closed-set.

Metrics are used to evaluate how well the model performs at its task. Common metrics for classification include *Accuracy* - the ratio of correct predictions to total predictions, *Precision* - the number of correct positive results divided by the total number of positive predictions, *Recall* (Sensitivity) - the number of correct positive results divided by the number of predictions that should have been positive.

For a given model there will be a tradeoff between Precision and Recall. For binary classification, the range of possible tradeoffs can be evaluated using a Receiver-Response Curve (ROC).

2.1.2 Training process

The goal of the classification model is to make good predictions *on unseen data*. The samples available in the dataset only represent some particular examples of this underlying (hidden) distribution of data. Care must be taken to avoid learning peculiarities that are specific to the training samples and not representative of general patterns. A model that fails this generalization criteria is often said to be *overfitting*, while a model that fails to learn any predictive patterns is said to be *underfitting*.

To address this challenge the dataset is divided into multiple subsets that have different purposes. The *training set* is data that the training algorithm uses to optimize the model on. To estimate how well the model generalizes to new unseen data, predictions are made on the *validation set*. The final performance of the trained model is evaluated on a *test set*, which has not been used in the training process. To get a better estimate of how the model performs K-fold cross-validation can be used, where K different training/validation splits are attempted. K is usually set to a value between 5 and 10. The overall process is illustrated in Figure 2.1.

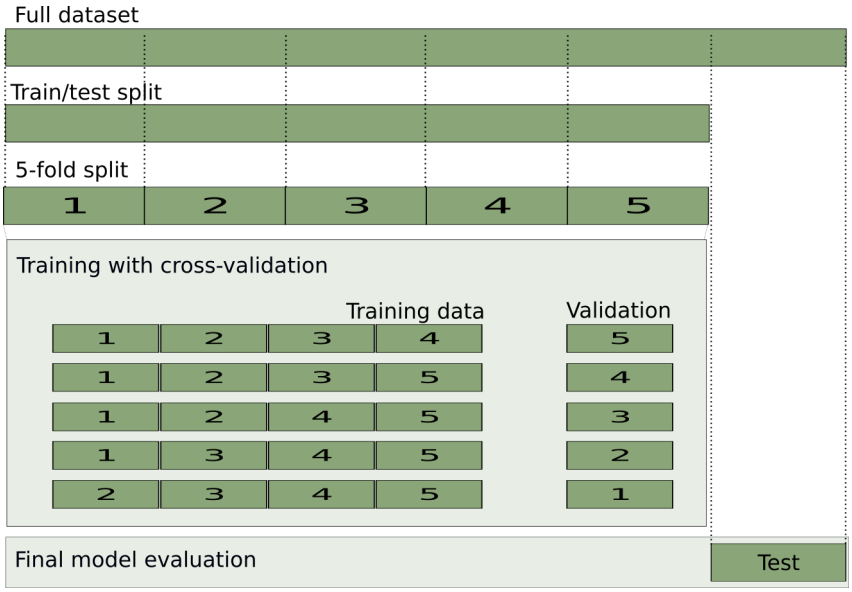


Figure 2.1: Splitting datasets into train/validation/test sets and cross-validation

One common style of supervised learning processes is to: start with a base model and initialize its parameters (often randomly), then make predictions using this model, compare these predictions with the labels to compute an error, and then update the parameters in order to attempt to reduce this error. This iterative process is illustrated in Figure 2.2.

Hyperparameters are settings for the training process. Hyperparameters can be chosen by trying different candidate settings, training model(s) to completion with these settings, and evaluating performance on the validation set. When performed systematically this is known as a hyperparameter search.

Once training is completed, the predictive model with the learned parameters can be used on new data.

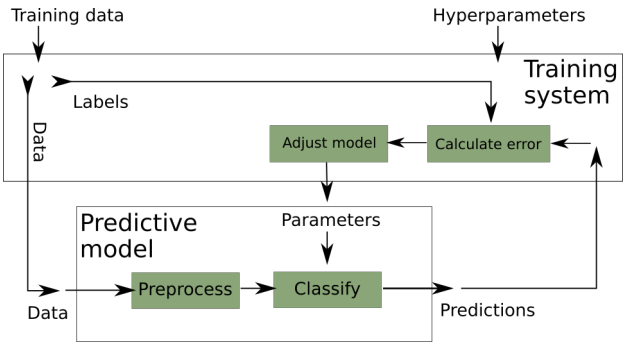


Figure 2.2: Relationship between training system and the predictive model during training

2.2 Neural Networks

Artificial Neural Networks are a family of machine learning methods, loosely inspired by the biological neurons in the brains of humans and other animals. Some of the foundations such as the Perceptron[20] dates back to the 1950s, but it was not until around 2010 that neural networks started to become the preferred choice for many machine learning applications.

2.2.1 Multi-Layer Perceptron

A basic and illustrative type of Neural Network is the Multi-Layer Perceptron (MLP), shown in Figure 2.3. It consists of an input layer, one or more hidden layers, and an output layer.

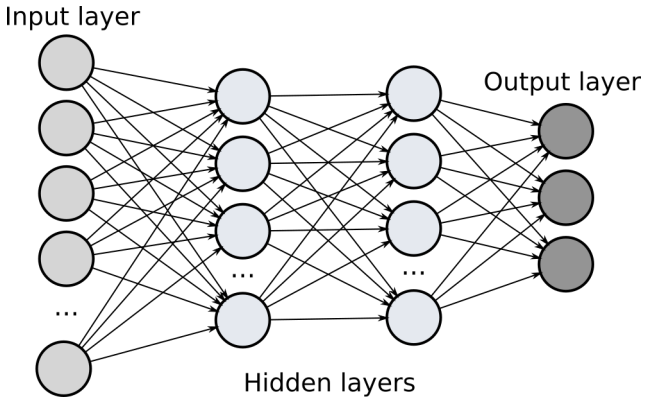


Figure 2.3: Multi-Layer Perceptron with 2 hidden layers

Each layer consists of a number of neurons. The neurons of one layer are connected to each of the neurons in the preceding layer. This type of layer is therefore known as a fully-connected, or densely-connected layer. The input to the network is a 1-dimensional vector. If the data is multi-dimensional (like an image) is to be used, it must be flattened to a 1-D vector.

Each neuron computes its output as a weighted sum of the inputs, offset by a bias and followed by an activation function f, as illustrated in 2.4. In the simplest case, the activation function is the identity function. This lets the layer express any linear function.

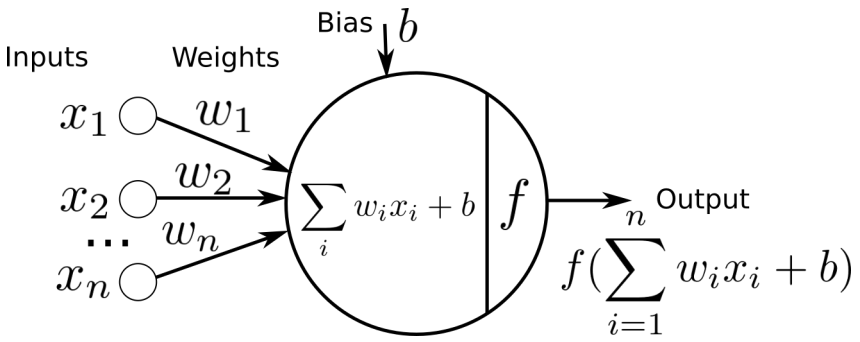


Figure 2.4: Computational principle of an artificial neuron

Making predictions with a neural network is done by applying the data as inputs to the first layer, then computing all the following layers until the final outputs. This is often called *forward propagation*.

2.2.2 Activation functions

To be able to express non-linear relationships between input and output, non-linear activation functions are applied. When non-linearity is used, a neural network becomes a universal function approximator[21].

Commonly used general-purpose non-linear activation functions are Tanh and ReLu[22]. Sigmoid and softmax are commonly used at the output stage of a neural network for classification, as they convert the input to a probability-like $(0, 1)$ range. Sigmoid is used for binary classification and Softmax for multi-class classification. To get a discrete class from these continuous probability values, a decision function is applied. The simplest decision function for single-label multi-class classification is to take the largest value, using the argmax function.

An illustration of the mentioned activation functions can be seen in Figure 2.5.

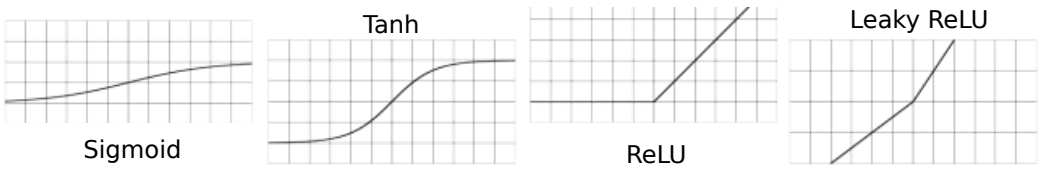


Figure 2.5: Commonly used activation functions in neural networks. Input shown along X-axis, output along Y. Range for Sigmoid is (0,1) and for Tanh (-1,1).

Increasing the number of neurons and the number of hidden layers increases the capacity of the network to learn more complex functions.

2.2.3 Training Neural Networks

Neural Networks are trained through numerical optimization of an objective function (loss function). For supervised learning, the standard method is mini-batch Gradient Descent with Backpropagation.

For classification, the cross-entropy (log loss) function is often applied. As the predicted probability of the true class gets close to zero, the log-loss goes towards infinity. This penalizes wrong predictions heavily, see Figure 2.6.

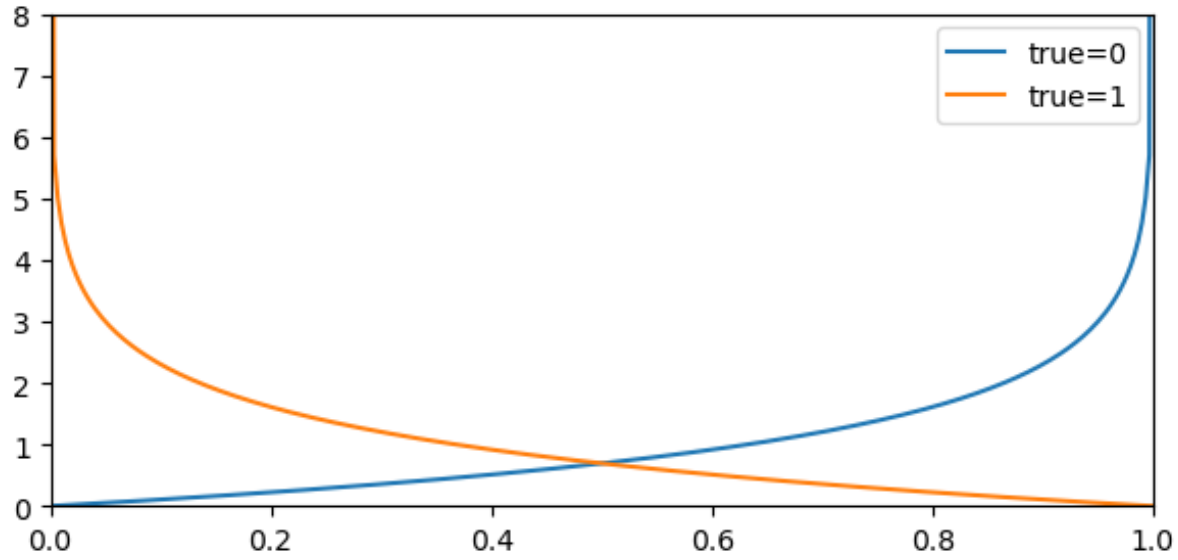


Figure 2.6: Plot of log-loss for binary classification.

Categorical cross-entropy is an extension of binary cross-entropy to multiple classes. Other loss functions are Logistic Loss, Mean Squared Error and Mean Absolute Error.

Making predictions with a forward pass of the neural network and computing the loss function allows estimating how well or how poorly the current model performs. In order to find out how the model parameters should be changed in order to perform better, *Gradient Descent* is applied. The gradient of a function expresses how much and in which direction its output varies with small changes to its inputs. This is computed as the partial derivative of the function.

The key to calculating the gradients in a multi-layer neural network is *backpropagation*[23]. Backpropagation works by propagating the error in the last layers "back" towards the input, using the partial derivative with respect to the inputs from the layer before it. This makes it possible to compute the gradients for each of the weights (and biases) in the network[24]. Once the gradients are known, the weights are updated by taking a step in the negative direction of the gradient. The step is kept small by multiplying with the *learning rate*, a hyperparameter in the range of 10^{-7} to 10^{-2}. Too small learning rates can lead to the model getting stuck in bad minima, while too large learning rates can cause training to not converge[24].

In *mini-batch* Gradient Descent, the training data is processed in multiple fixed-size batches of data, and the loss function and model parameters updates are computed per

batch. This means that not all the training data has to be kept in memory at the same time, which allows training on very large datasets. The batch size is a hyperparameter and has to be set high enough for the batch loss to be a reasonable estimate of the loss on the full training set, but small enough for the batch to fit into memory.

One pass through the entire training set is called an *epoch*, and training normally consists of many epochs.

The mini-batch Gradient Descent optimization with backpropagation can be summarized in the following procedure:

1. Sample a mini-batch of data
2. Forward propagate the data to compute output probabilities, calculate the loss
3. Backpropagate the errors to compute error gradients in the entire network
4. Update each weight by moving a small amount against the gradient
5. Go to 1) until all batches of all epochs are completed

Gradient Descent is not guaranteed to find a globally optimal minimum, but with suitable choices of hyperparameters can normally find local minima that are good-enough. It has also been argued that a global optimum on the training set might not be desirable, as it is unlikely to generalize well[25].

2.2.4 Convolutional layers

Convolutional layers are an alternative to fully-connected layers that allows for 2D input and that can exploit the spatial relationship in such data.

In addition to the width and height, the input also has a third dimension, the number of *channels*. The different channels can contain arbitrary kinds of data, but common for the first layer would be 3 channels with an RGB color image or 1 channel for grayscale images or audio spectrogram.

To perform 2D convolution, a filter (or *kernel*) of fixed size K_wxK_h is swept across the 2D input of a channel. For each location, the kernel multiplies the input data with the *kernel weights* and sums to a single value that becomes the output value. This is shown in Figure 2.7.

This convolution process allows expressing many useful transformations on 2D data, by simply selecting different weights in the kernel. Simple examples are edge detection (horizontal/vertical/diagonal) or smoothening filters (mean/Gaussian). And since the kernels weights in a convolutional layer are trained, they learn to detect local features specific to the training data.

With multiple input channels, the same kernel is applied over all the input channels, and all the results at one location, and the bias, are summed together to become the output. Multiple convolution filters are normally used per layer, to produce M new output channels with different features from the N input channels.

In Figure 2.7 each location of the kernel has the entire kernel inside the input area. This is called convolution with "valid" padding, and the resulting output will be smaller by $\lceil k/2 \rceil$ on each side, where k is the kernel size. If the input is instead padded by this amount when moving the kernel, the output will be the same size as the input. This is called convolution with "full" padding.

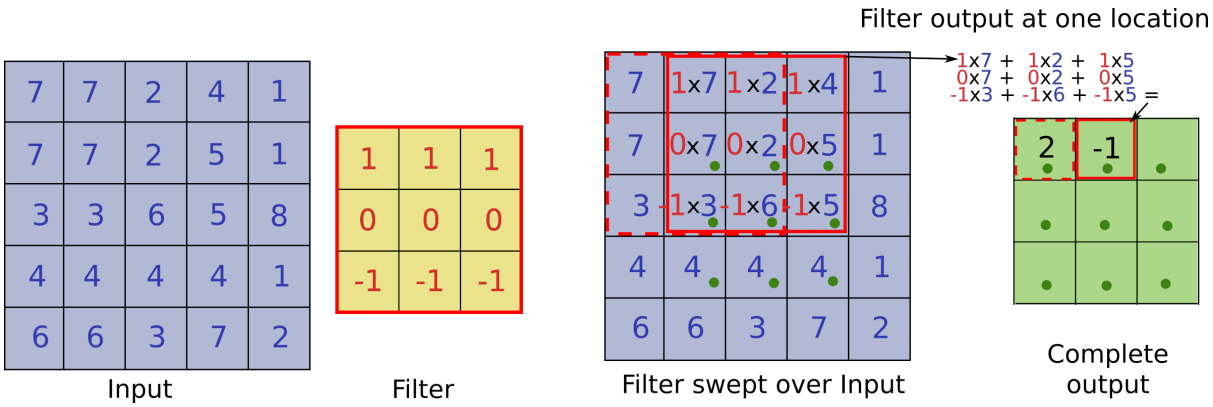


Figure 2.7: 2D convolution for a single channel. Red outlines show how the filter moves across the input image. Filter weights shown in red numbers, input numbers in blue. Green dots illustrate locations of outputs with respect to inputs.

The number of learnable parameters in a 2D convolutional layer with bias is $P_{conv} = M(K_w K_h + 1)$, with M filters and a kernel of size K_wxK_h. Commonly used kernel sizes are 3x3 to 7x7.

In the fully-connected layer, there were $(N + 1) \times M$ parameters for N inputs and M neurons. For 2D images $N = height \times width \times channels$, so even for small image grayscale inputs (N=100x100x1=10000) a convolutional layer has much fewer parameters.

The computational complexity of a Neural Network is often measured in the number of Multiply-Accumulate operations (MACC).

The computational complexity of a 2D convolution is $O_{conv} = WHNK_w k_h M$, with input height H, input width W, N input channels, M output channels$ and a 2D kernel of size K_wxK_h.

2.2.5 Convolutional Neural Network

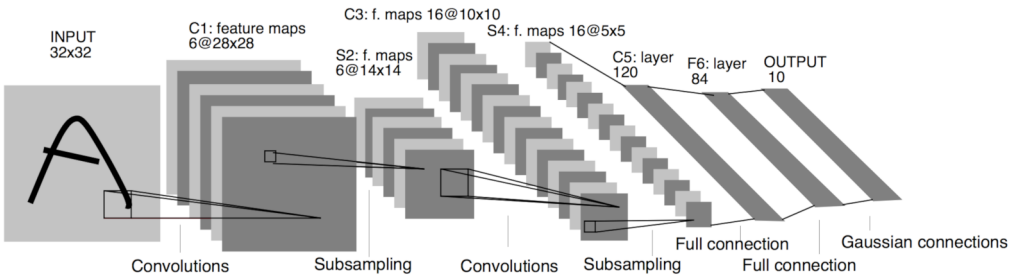


Figure 2.8: The LeNet-5 architecture illustrated. From the original paper[26]

A Convolutional Neural Network (CNN) is a neural network that uses convolutional layers in addition to (or instead of) fully-connected layers. One of the early examples of a CNN model was LeNet5 (1998)[26], which was successfully applied to the recognition of handwritten digits. As seen in Figure 2.8, the architecture uses two convolutional layers (with subsampling after each), followed by two fully-connected layers and then the output layer.

Architectures with more layers based on very similar structures have been shown to work well also on more complex tasks, like VGGNet (2014)[27] on the 1000-class image recognition task ImageNet[28].

2.2.6 Subsampling

Besides the convolution filters, the other critical part that makes CNNs effective is to gradually subsample the data as it moves through the convolutional layers. This forces the model to learn bigger (relative to the original input space) and more complex features (patterns of patterns) in later layers.

A pooling layer is one way to accomplish this, and was used in the LeNet5 and VGGNet architectures. A 2D pooling operation has K_wxK_h sized filter and scans over the image width and height. The stride is normally set to the same size of the operation. Each channel is processed independently. It outputs 1 element for each scanned location in the image. With *average pooling*, the output is the average value of the input. With *max pooling*, the output is the maximum value of the input (Figure 2.9).

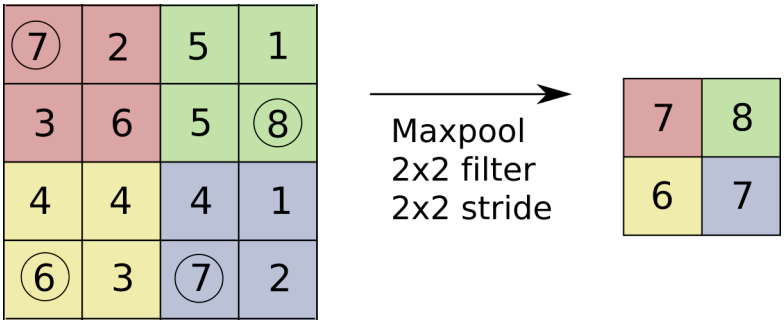


Figure 2.9: Max pooling operation. Different positions of the filter are colorized, with the maximum value in each position circled.

Another way of subsampling is by increasing the stride of the convolutions[29]. If a kernel has a stride of 2 (as in Figure 2.10), then the output of the convolution will be reduced by half. Striding is usually applied in the first convolution in a layer and reduces the number of computations compared to pooling because fewer inputs need to be computed.

Striding was used to replace most of the pooling operations in ResNet[30] (2015), which beat human-level performance on the ImageNet task.

2.2.7 Spatially Separable convolution

While CNNs have a relatively low number of parameters, compared to fully-connected layers, the convolutional kernels still require a lot of computations. Several alternative convolution layers have been proposed to address this.

In a spatially separable convolution, a 2D convolution is factorized into two convolutions with 1D kernels. First, a 2D convolution with a $1xK_h$ kernel is performed, followed by a 2D convolution with a K_wx1 kernel, as illustrated in Figure 2.11. The number of operations is

$$O_{ss} = HWNMK_w + HWNMK_h = HWNM(K_w + K_h)$$

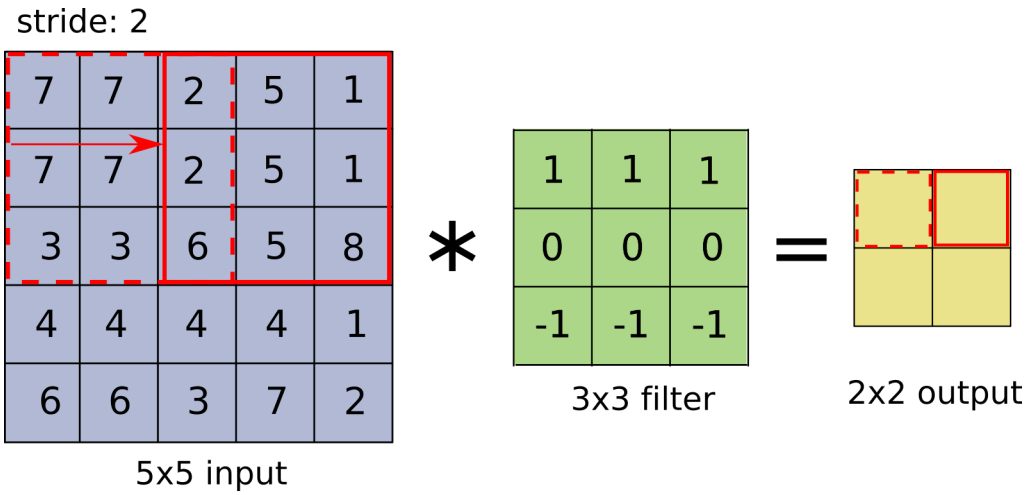


Figure 2.10: Strided convolution. The kernel input (marked in red) moves by stride=2, effectively subsampling the input image

The number of computations and parameters compared with regular 2D convolutions is $(K_w K_h)/(K_w + K_h)$ times fewer. For example with $K_w = K_h = 3$, $9/6 = 1.5$ and with $K_w = K_h = 5 = 25/10 = 2.5$.

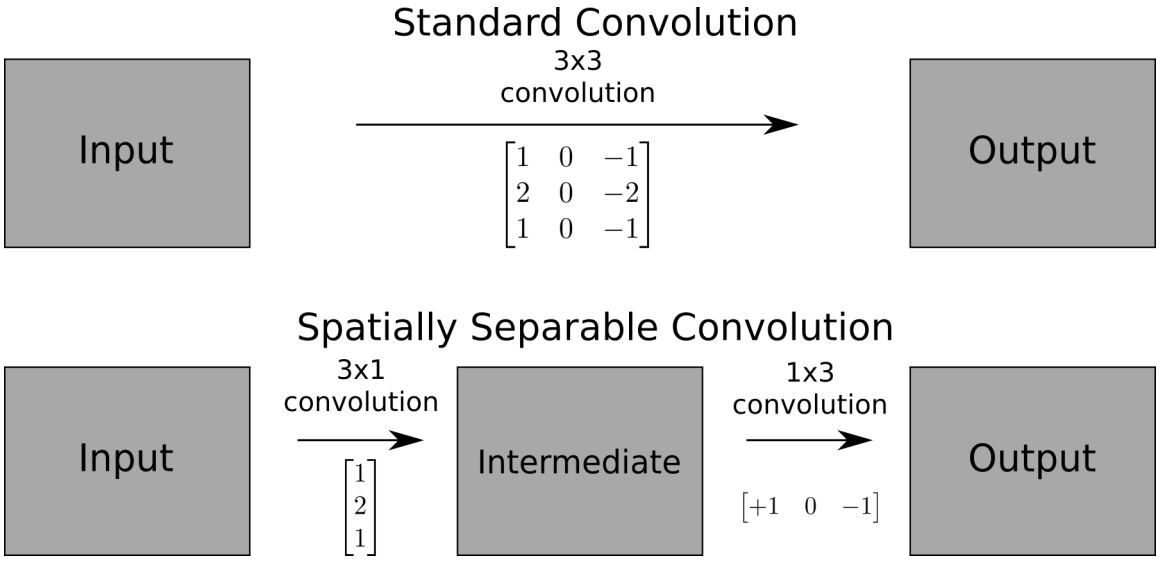


Figure 2.11: Spatially Separable 2D convolution versus standard 2D convolution

2.2.8 Depthwise Separable convolution

While a standard convolutional layer performs a convolution over both channels and the spatial extent a Depthwise Separable convolution splits this into two convolutions: First a *Depthwise Convolution* over the spatial extent only, followed by a *Pointwise Convolution* over the input channels. The difference is illustrated in Figure 2.12.

The pointwise convolution is sometimes called a 1x1 convolution since it is equivalent to a 2D convolution operation with a 1x1 kernel.

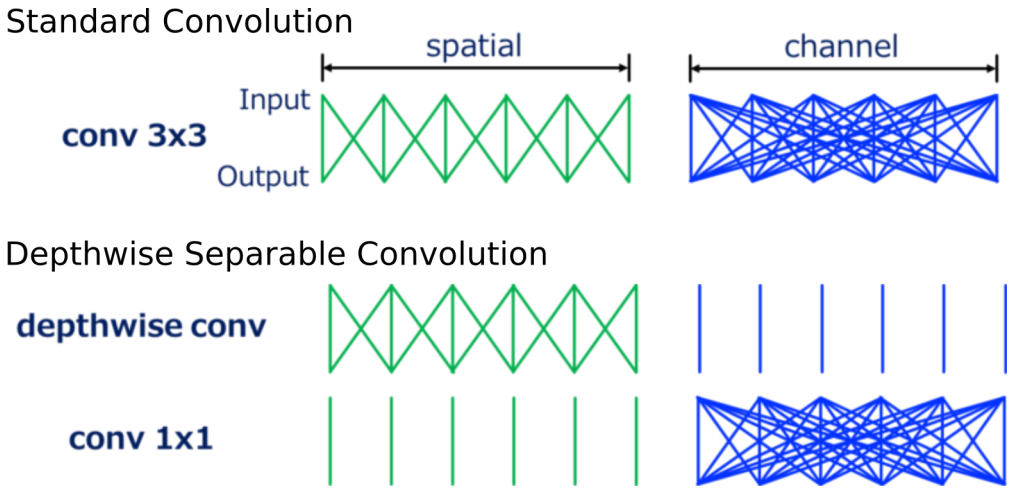


Figure 2.12: Input/output relationship of standard 3x3 convolution versus Depthwise Separable convolution. The image is based on illustrations by Yusuke Uchida[31]

$$O_{pw} = HWNM$$
$$O_{dw} = HWNK_wK_h$$
$$O_{ds} = O_{pw} + O_{dw} = HWN(M + K_wK_h)$$

This factorization requires considerably fewer computations compared to full 2D convolutions. For example, with kernels size $K_w = K_h = 3$ and $M = 64$ channels, it takes approximately $7.5x$ fewer operations.

2.2.9 Efficient CNNs for Image Classification

The development of more efficient Convolutional Neural Networks for image classification has received a lot of attention over the last few years. This is especially motivated by the ability to run models that give close to state-of-the-art performance on mobile phones and tablets. Since spectrograms are 2D inputs that are similar to images, it is possible that some of these techniques can transfer over to Environmental Sound Classification.

SqueezeNet[32] (2015) focused on reducing the size of model parameters. It demonstrated AlexNet[33]-level accuracy on ImageNet challenge using 50x fewer parameters, and the parameters can be compressed to under 0.5MB in size compared to 240MB for AlexNet. It replaced most 3x3 convolutions in a convolution block with 1x1 convolutions, and reduce the number of channels using "Squeeze" layers consisting only of 1x1 convolutions. The paper also found that a residual connection between blocks increased model performance by 2.9% without adding parameters.

Mobilenets[34] (2017) focused on reducing inference computations by using Depthwise separable convolutions. A family of models with different complexity was created using two hyperparameters: a width multiplier α (0.0-1.0) which adjusts the number of filters in each convolutional layer, and the input image size. On ImageNet, MobileNet-160 $\alpha = 0.5$ with 76M MACC performs better than SqueezeNet with 1700M MACC, a 22x reduction. The smallest tested model was 0.25 MobileNet-128, with 15M MACC and 200k parameters.

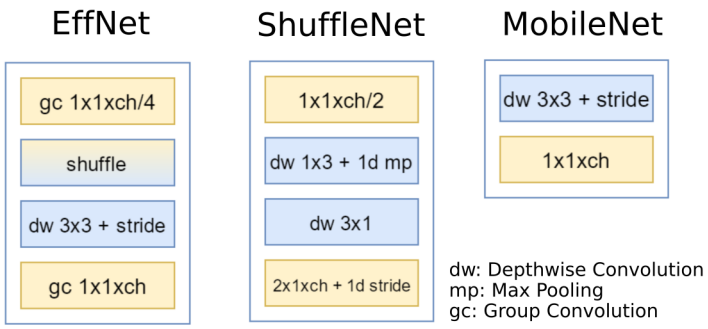


Figure 2.13: Convolutional blocks of Effnet, ShuffleNet, and Mobilenet. Illustration based on Effnet paper[35]

Shufflenet[36] (2017) uses group convolutions in order to reduce computations. In order to mix information between different groups of convolutions, it introduces a random channel shuffle.

SqueezeNext[37] (2018) is based on SqueezeNet but uses spatially separable convolution (1x3 and 3x1) to improve inference time. While the MACC count was higher than MobileNet, they claim better inference time and power consumption on their simulated hardware accelerator.

Effnet[35] (2018) also uses spatial separable convolutions, but additionally performs the downsampling in a separable fashion: first a 1x2 max pooling after the 1x3 kernel, followed by 2x1 striding in the 3x1 kernel. Evaluated on CIFAR10 and Street View House Numbers (SVHN) datasets it scored a bit better than Mobilenets and ShuffleNet.

2.3 Audio Classification

In Audio Classification, the predictive models operate on audio (digital sound). Example tasks are wake-word or speech command detection in Speech Recognition, music genre or artist classification in Music Information Retrieval - and classification of environmental sounds.

2.3.1 Digital sound

Physically, sound is a variation in pressure over time. To process the sound with machine learning, it must be converted to a digital format. The acoustic data is first converted to analog electric signals by a microphone and then digitized using an Analog-to-Digital-Converter (ADC), as illustrated in Figure 2.14.

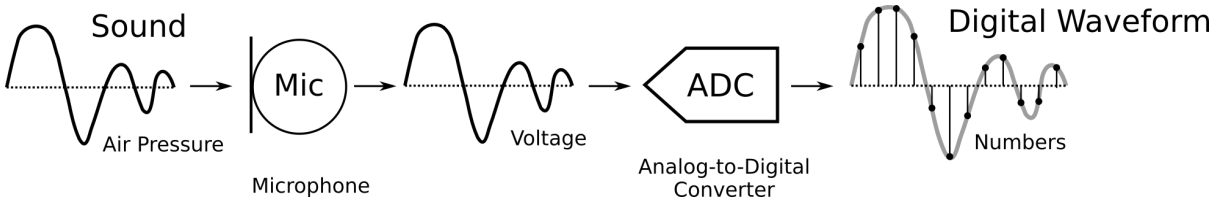


Figure 2.14: Conversion of sound into a digital representation

In the digitization process, the signal is quantized in time at a certain sampling frequency, and the amplitude quantized at a certain bit-depth. A typical sampling frequency is 44100 Hz and a bit-depth 16 bit, as used in the Audio CD format[38]. With such parameters, the majority of human-perceivable information in the acoustic sound is captured. In this representation sound is a one-dimensional sequence of numbers, sometimes referred to as a *waveform*. This is the format utilized by case A) in Figure {figure:sensornetworks-coding} from the introduction.

Digital sound can be stored uncompressed (example format: WAV PCM[39]), using lossless compression (FLAC[40]) or using lossy compression (MP3[41]). Lossy compression removes information that is indistinguishable to the human ear and can compress better than lossless. But lossy compression can add compression artifacts, and is best avoided for machine learning tasks.

Recordings can have multiple channels of audio but for machine learning on audio single-channel data (mono-aural) is still common.

2.3.2 Spectrogram

Sounds of interest often have characteristic patterns not just in time (temporal signature) but also in frequency content (spectral signature). Therefore it is common to analyze audio in a time-frequency representation (a *spectrogram*).

A common way to compute a spectrogram from an audio waveform is by using the Short-Time Fourier Transform (STFT)[42]. The STFT operates by splitting the audio up in short consecutive chunks and computing the Fast Fourier Transform (FFT) to estimate the frequency content for each chunk. To reduce artifacts at the boundary of chunks, they

are overlapped (typically by 50%) and a window function (such as the Hann window) is applied before computing the FFT. With the appropriate choice of window function and overlap, the STFT is invertible[43].

There is a trade-off between frequency (spectral) resolution and time resolution with the STFT. The longer the FFT window the better the frequency resolution, but the poorer the temporal resolution. For speech, a typical choice of window length is 20 ms. Similar frame lengths are often adopted for acoustic events. The STFT returns complex numbers describing the phase and magnitude of each frequency bin. A spectrogram is computed by squaring the absolute of the magnitude and discarding the phase information. This is called a *linear spectrogram* or sometimes just spectrogram. The lack of phase information means that the spectrogram is not strictly invertible, though estimations exist[44][45]. A linear spectrogram can be on top in Figure **??**.

2.3.3 Mel-spectrogram

The more complex the input to a machine learning system is, the more processing power is needed both for training and for inference. Therefore one would like to reduce the dimensions of inputs as much as possible. A linear spectrogram often has considerable correlation (redundant information) between adjacent frequency bins and is often reduced to 30-128 frequency bands using a filter-bank. Several different filter-bank alternatives have been investigated for audio classification tasks, such as 1/3 octave bands, the Bark scale, Gammatone, and the Mel scale. All these have filters spacing that increases with frequency, mimicking the human auditory system. See Figure 2.15.

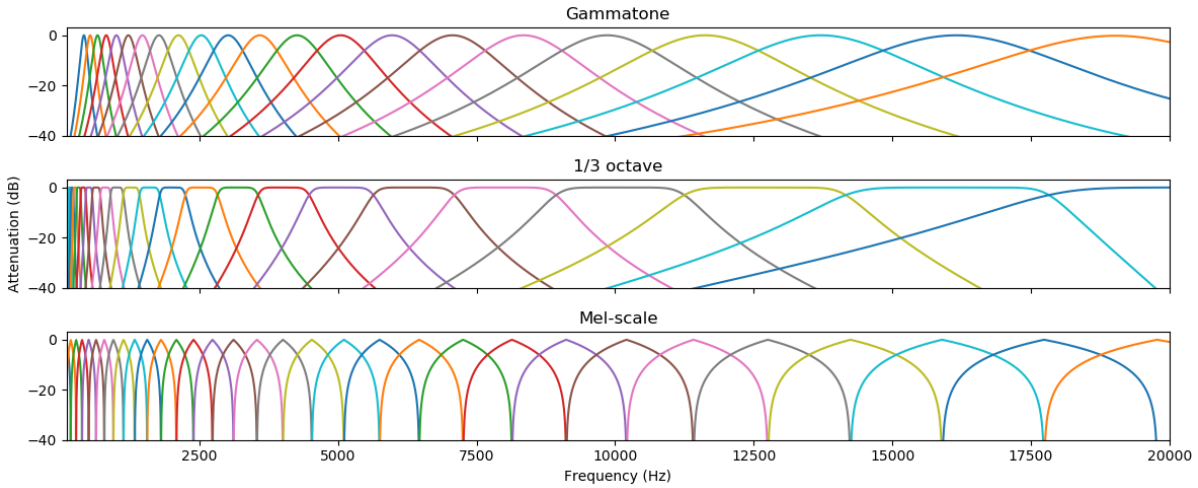


Figure 2.15: Comparison of different filterbank responses: Mel, Gammatone, 1/3-octave.

The Mel scaled filters are commonly used for audio classification[46]. The spectrogram that results for applying a Mel-scale filter-bank is often called a Mel-spectrogram.

Mel-Filter Cepstral Coefficients (MFCC) is a feature representation computed by performing a Discrete Cosine Transform (DCT) on a mel-spectrogram. This further reduces dimensionality to just 13-20 bands with low correlation between each band. MFCC features have been very popular in speech recognition tasks[46], however in general sound classification tasks mel-spectrograms tend to perform better[47][48].

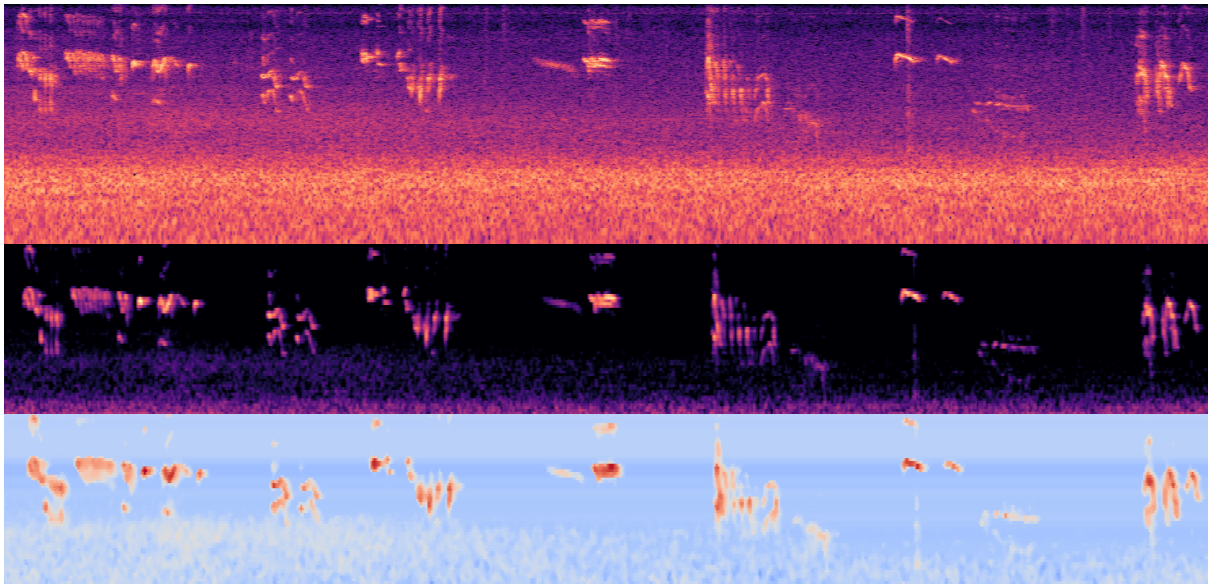


Figure 2.16: Different spectrograms showing birdsong. Top: Linear spectrogram. Middle: Mel-spectrogram. Bottom: Normalized mel-spectrogram after mean-subtraction and standard scaling. The Mel-spectrograms in this example had the first filter set to 1kHz, eliminating a lot of the low-frequency noise seen in the linear spectrogram.

2.3.4 Normalization

Audio has a very large dynamic range. The human hearing has a lower threshold of hearing down to 20μPa (0 dB SPL) and a pain threshold of over 20 Pa (120 dB SPL), a difference of 6 orders of magnitude[49, Ch. 22]. A normal conversation might be 60 dB SPL and a pneumatic drill 110 dB SPL, 4 orders of magnitude difference. It is common to compress the range of values in spectrograms by applying a log transform.

In order to center the values, the mean (or median) of the spectrogram can be removed. Scaling the output to a range of 0-1 or -1,1 is also sometimes done. These changes have the effect of removing amplitude variations, forcing the model to focus on the patterns of the sound regardless of amplitude.

2.3.5 Analysis windows

When recording sound, it forms a continuous, never-ending stream of data. The machine learning classifier however generally needs a fixed-size feature vector. Also when playing back a finite-length recording, the file may be many times longer than the sounds that are of interest. To solve these problems, the audio stream is split up into analysis windows of fixed length, which are classified independently. The length of the window is typically a little longer than the longest target sound. The windows can follow each-other with no overlap, or move forward by a number less than the window length (overlap). With overlap, a target sound will be classified a couple of times, at a slightly different position inside the analysis window each time. This can improve classification accuracy.

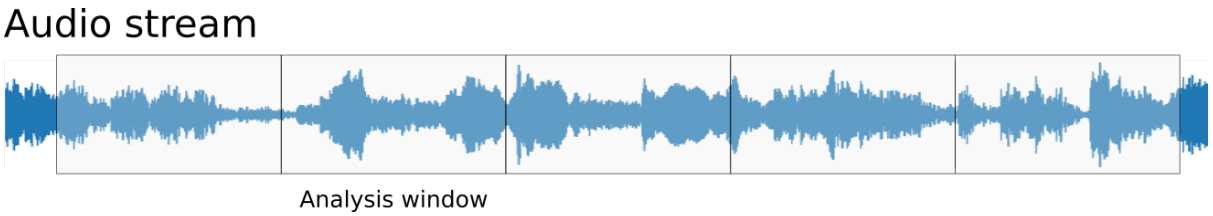


Figure 2.17: Audio stream split into fixed-length analysis windows without overlap

A short analysis window has the benefit of reducing the feature size of the classifier, which uses less memory and possibly allows to reduce the model complexity, and in turn allow to make better use of a limited dataset.

When the length of audio clips is not evenly divisible by length of analysis windows, the last window is zero-padded.

2.3.6 Weak labeling

Sometimes there is a mismatch between the desired length of the analysis window, and the labeled clips available in the training data. For example, a dataset may consist of labeled audio clips with a length of 10 seconds, while the desired output is every second. When a dataset is labeled only with the presence of a sound at a coarse timescale, without information about where exactly the relevant sound(s) appears it is referred to as *weakly annotated* or *weakly labeled* data[50].

If one assumes that the sound of interest occurs throughout the entire audio clip, a simple solution is to let each analysis window inherit the label of the audio clip without modification, and to train on individual analysis windows. If this assumption is problematic, the task can be approached as a Multiple Instance Learning (MIL) problem. Under MIL each training sample is a bag of instances (in this case, all analysis windows in an audio clip), and the label is associated with this bag[51]. The model is then supposed to learn the relationship between individual instances and the label. Several MIL techniques have been explored for audio classification and audio event detection[52][53][54].

2.3.7 Aggregating analysis windows

When evaluating a test-set where audio clips are 10 seconds, but the model classifies analysis windows of 1 second the individual predictions must be aggregated into one prediction for the clip.

A simple technique to achieve this is *majority voting*, where the overall prediction is the class that occurs most often across individual predictions.

With *soft voting* or *probabilistic voting*, the probabilities of individual predictions are averaged together, and the output prediction is the class with the highest probability overall.

2.3.8 Data augmentation

Access to labeled samples is often limited because they are expensive to acquire. This can be a limiting factor for reaching good performance using supervised machine learning.

Data Augmentation is a way to synthetically generate new labeled samples from existing ones, in order to expand the effective training set. A simple form of data augmentation can be done by modifying the sample data slightly. Common data augmentation techniques for audio include Time-shift, Pitch-shift, and Time-stretch. These are demonstrated in Figure 2.18.

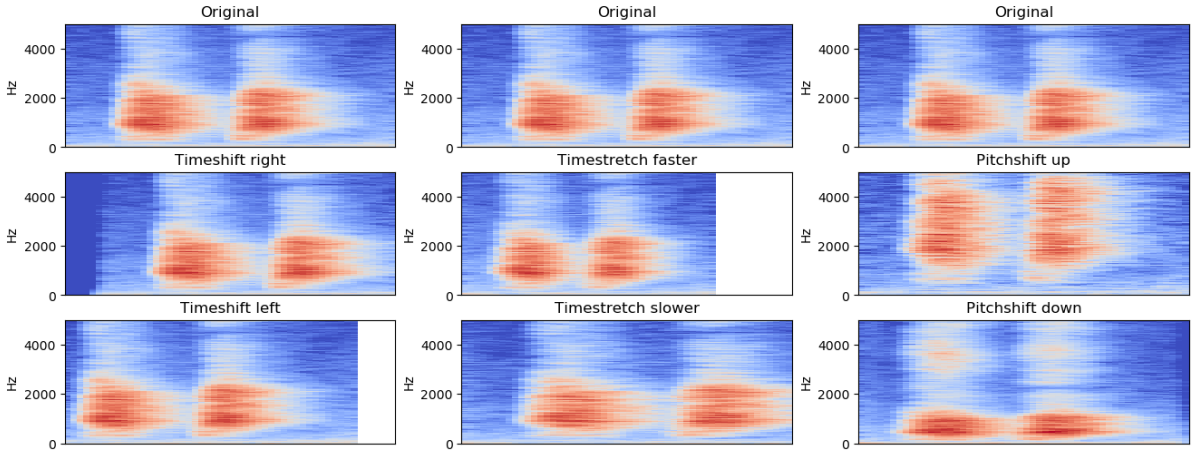


Figure 2.18: Common data augmentations for audio demonstrated on a dog bark ("woof woof"). Figure shows log-scaled linear spectrograms before and after applying the augmentation. Parameters are exaggerated to show the effects more clearly.

Mixup[55] is another type of data augmentation technique where two samples from different classes are mixed together to create a new sample. A mixup ratio λ controls how much the sample data is mixed, and the labels of the new sample are mixed in the same way.

$$\tilde{x} = \lambda x_i + (1-\lambda)x_j \qquad \text{where } x_i, x_j \text{are raw input vectors}$$
$$\tilde{y} = \lambda y_i + (1-\lambda)y_j \qquad \text{where } y_i, y_j \text{are labels one-hot encoded}$$

The authors argue that this encourages the model to behave linearly in-between training examples. It has been shown to increase performance on audio tasks[56][57][58].

Data augmentation can be applied either to the raw audio waveform or to preprocessed spectrograms.

2.3.9 Efficient CNNs for Speech Detection

Speech detection is a big application of Audio Classification. A lot of research has been focused on efficient models for use in smartwatches, mobile devices and smart-home devices (Amazon Alexa et.c.). In the Keyword Spotting (KWS) the goal is to detect a keyword or phrase that indicates that the user wants to enable speech control. Being an audio classification task that is often done on low-power microcontrollers, some of the explored techniques may transfer over to the Environmental Sound Classification task.

In [59] (2015) authors evaluated variations of small-footprints CNNs for keyword spotting. They found that using large strides in time or frequency could be used to create models that were significantly more computationally effective.

In the "Hello Edge"[60] paper (2017), different models were evaluated for keyword spotting on microcontrollers. Included were most standard deep learning model architectures such as Deep Neural Networks (DNN), Recurrent Neural Networks and Convolutional Neural Networks. They found that Depthwise Separable Convolutional Neural Network (DS-CNN) provided the best accuracy while requiring significantly lower memory and compute resources than other alternatives. Models were evaluated with three different performance limits. Their "Small" version with under 80KB, 6M ops/inference achieved 94.5% accuracy on the Google Speech Command dataset. A DNN version was demonstrated on a high-end microcontroller (ARM Cortex M7 at 216 Mhz) using CMSIS-NN framework, running keyword spotting at 10 inferences per second while utilizing only 12% CPU (rest sleeping).

FastGRNN[61] (2018) is a Gated Recurrent Neural Network designed for fast inference on audio tasks on microcontrollers. It uses a simplified gating architecture with residual connection and a three-stage training schedule that forces weights to be quantized in a sparse and low-rank fashion. When evaluated on Google Speech Command Set (12 classes), their smallest model of 5.5 KB achieved 92% accuracy and ran in 242 ms on a low-end microcontroller (ARM Cortex M0+ at 48 Mhz).

2.4 Environmental Sound Classification

Environmental Sound Classification, or Environmental Sound Recognition, is the task of classifying environmental sounds or noises. It has been researched actively within the machine learning community at least since 2006[62].

2.4.1 Datasets

The Urbansound taxonomy[63] is a proposed taxonomy of sound sources, developed based on analysis of noise complaints in New York City between 2010 and 2014. The same authors also compiled the Urbansound dataset[63], based on selecting and manually labeling content from the Freesound[64] repository. 10 different classes from the Urbansound taxonomy were selected and 1302 different recordings were annotated, for a total of 18.5 hours of labeled audio. A curated subset with 8732 audio clips with a maximum length of 4 seconds is known as *Urbansound8k*.

class	Samples	Duration (avg)	In foreground
air_conditioner	1000	3.99 s	56 %
car_horn	429	2.46 s	35 %
children_playing	1000	3.96 s	58 %
dog_bark	1000	3.15 s	64 %
drilling	1000	3.55 s	90 %
engine_idling	1000	3.94 s	91 %
gun_shot	374	1.65 s	81 %
jackhammer	1000	3.61 s	73 %
siren	929	3.91 s	28 %
street_music	1000	4.00 s	62 %

Table 2.1: Classes found in the Urbansound8k dataset

YorNoise[65] is a collection of vehicle noise. It has a total of 1527 samples, in two classes: road traffic (cars, trucks, buses) and rail (trains). The dataset follows the same design as Urbansound8k, and can be used standalone or as additional classes to Urbansound8k.

ESC-50[66] is a small dataset of environmental sounds, consisting of 2000 samples across 50 classes from 5 major categories. The dataset was compiled using sounds from Freesound[64] online repository. A subset of 10 classes is also proposed, often called ESC-10. Human accuracy was estimated to be to 81.30% on ESC-50 and 95.7% on ESC-10[66, Ch. 3.1]. The Github repository for ESC-50[67] contains a comprehensive summary of results on the dataset, with over 40 entries. As of April 2019, the best models achieve 86.50% accuracy, and all models with over 72% accuracy use some kind of Convolutional Neural Network.

AudioSet [17] is a large general-purpose ontology of sounds with 632 audio event classes. The accompanying dataset has over 2 million annotated clips based on audio from Youtube videos. Each clip is 10 seconds long. 527 classes from the ontology are covered.

In the DCASE2019 challenge (in progress, ends July 2019) task 5[68] audio clips containing common noise categories are to be tagged. The tagging is formulated as a multi-label classification on 10-second clips. The dataset[69] has 23 fine-grained classes across 8 categories with 2794 samples total. The data was collected from the SONYC noise sensor network in New York City.

Several earlier DCASE challenge tasks and datasets have been on related topics to Environmental Sound Classification, such as Acoustic Scene Detection[70], general-purpose tagging of sounds[71], and detection of vehicle-related sounds[72].

2.4.2 Spectrogram-based models

Many papers have used Convolutional Neural Networks (CNN) for Environmental Sound Classification. Approaches based on spectrograms and in particular log-scaled Mel-spectrogram being the most common.

PiczakCNN[73] in 2015 was one of the first applications of CNNs to the Urbansound8k dataset. It uses 2 channels of log-Mel-spectrograms, both the plain spectrogram values and the first-order difference (delta spectrogram). The model uses 2 convolutional layers, first with size 57x6 (frequency x time) and then 1x3, followed by two fully connected layers with 5000 neurons each. The paper evaluates short (950 ms) versus long (2.3 seconds) analysis windows and majority voting versus probability voting. Performance on Urbansound8k ranged from 69% to 73%. It was found that probability voting and long windows perform slightly better[73].

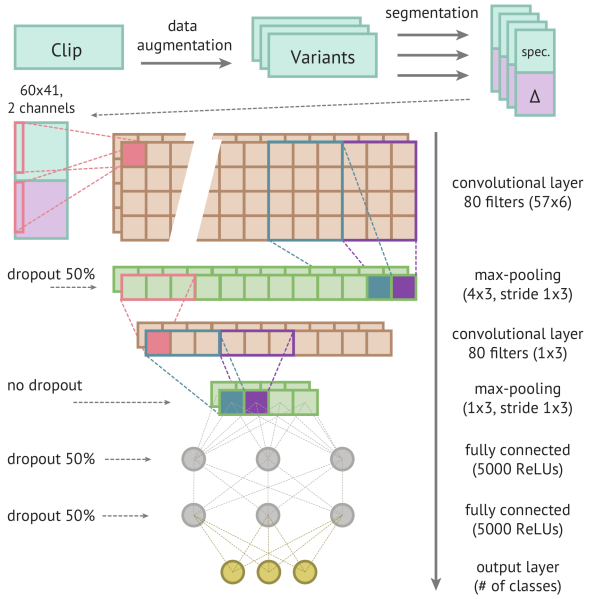


Figure 2.19: The architecture of Piczak CNN, from the original paper [73]. The model input has two channels: the spectrogram magnitude (light blue) and a first-order difference "delta" of the spectrogram (purple))

SB-CNN[74] (2016) is a 3-layer convolutional with uniform 5x5 kernels and 4x2 max pooling. The paper also analyzes the effects of several types of data augmentation on Urbansound8k. including Time Shift, Pitch Shift, Dynamic Range Compression and Background Noise. With all augmentations, performance on their model raised from 72% to 79% classification accuracy. However time-stretching and pitch-shifting were the only techniques that consistently gave a performance boost across all classes.

D-CNN[75] (2017) uses feature representation and model architecture that largely follows that of PiczakCNN, however, the second layer uses dilated convolutions with a dilation

rate of 2. With additional data augmentation of time-stretching and noise addition, this gave a performance of up to 81.9% accuracy on Urbansound8k. LeakyRelu was found to perform slightly better than ReLu which scored 81.2%.

A recent paper investigated the effects of mixup for data augmentation[56]. Their model uses 4 blocks of 2 convolutional layers each, with each block followed by max pooling. The second block and third block together form a spatially separated convolution: the second block uses two 3x1 convolutions, and third block uses two 1x5 convolutions. On Mel-spectrograms the model scored 74.7% on Urbansound8k without data augmentation, 77.3% with only mixup applied, and 82.6% when time stretching and pitch shift was combined with mixup. When using Gammatone spectrogram features instead of Mel-spectrogram performance increased to 83.7%, which seems to be state-of-the-art as of April 2019.

2.4.3 Audio waveform models

Recently approaches that use the raw audio waveform as input have also been documented.

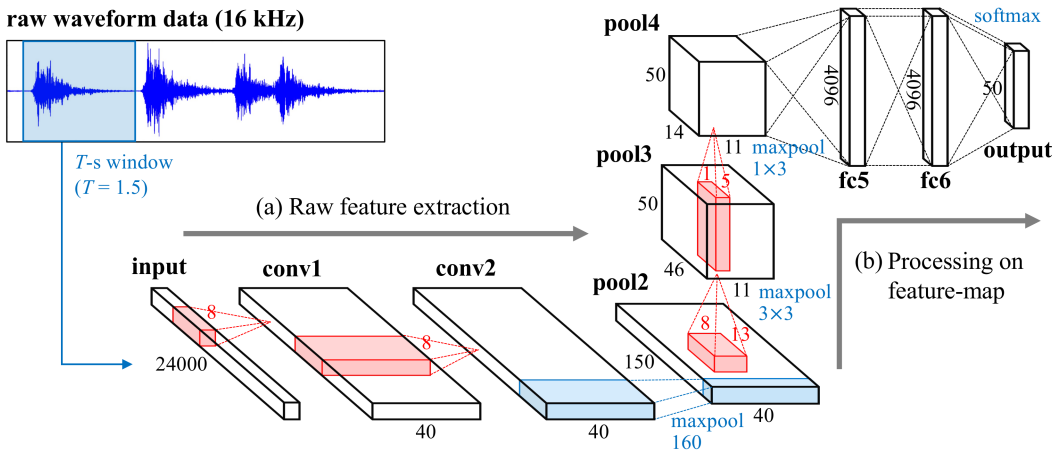


Figure 2.20: EnvNet[76] architecture, using raw audio as input.

EnvNet[76] (2017) used 1D convolutions in order to learn a 2D spectrogram-like representation which is then classified using standard 2D convolutional layers. The architecture is illustrated in Figure 2.20. They show that the resulting spectrograms have frequency responses with a shape similar to mel-spectrograms. The model achieves a 66.3% accuracy score on Urbansound8k[77] with raw audio input.

In [78], the authors evaluated a number of deep CNNs using only 1D convolutions. Raw audio with 8 kHz sample rate was used as the input. Their 18 layer model (M18) got a 71% accuracy on Urbansound8k, and the 11 layer version (M11) got 69%.

EnvNet2[77] (2018) is like EnvNet but with 13 layers total instead of 7, and using 44.1 kHz input sample-rate instead of 16 kHz. Without data augmentation, it achieves 69.1% accuracy on Urbansound8k. When combining data augmentation with Between-Class examples, a technique similar to Mixup, the model is able to reach 78.3% on Urbansound8k.

2.4.4 Resource-efficient models

There are also a few studies on Environmental Sound Classification (ESC) that explicitly target making resource-efficient models, measured in number of parameters and compute operations.

WSNet[79] is a 1D network on raw audio designed for efficiency. It proposes a weight sampling approach for efficient quantization of weights to reach an accuracy of 70.5% on UrbandSound8k with 288 K parameters and 100 M MAC.

LD-CNN[80] is a more efficient version of D-CNN. In order to reduce parameters the early layers use spatially separable convolutions, and the middle layers used dilated convolutions. As a result, the model has 2.05MB of parameters, 50x fewer than D-CNN, while accuracy only dropped by 2% to 79% on Urbansound8k.

AclNet [57] is an end-to-end CNN architecture. It uses 2 layers of 1D strided convolution as a learned filterbank to create a 2D spectrogram-like set of features. Then a VGG style architecture with Depthwise Separable Convolutions is applied. A width multiplier allows to adjust model complexity by changing the number of kernels in each layer, and a number of model variations were tested. Data augmentation and mixup was applied, and gave up to a 5% boost. Evaluated on ESC-50, the best performing model gets 85.65% accuracy, very close to state-of-the-art. The smallest model had 7.3 M MACC with 15 k parameters and got 75% accuracy on ESC-50.

eGRU[81] demonstrates a Recurrent Neural Network based on a modified Gated Recurrent Unit. The feature representation used was raw STFT spectrogram from 8Khz audio. The model was tested using Urbansound8k, however it did not use the pre-existing folds and test-set, so the results may not be directly comparable to others. With full-precision floating-point the model got 72% accuracy. When running on device using the proposed quantization technique the accuracy fell to 61%.

As of April 2019, eGRU was the only paper that could be found for the ESC task and the Urbansound8k dataset on a microcontroller.

2.5 Microcontrollers

A microcontroller is a tiny computer integrated on a single chip, containing CPU, RAM, persistent storage (FLASH) as well as peripherals for communicating with the outside world.

Common forms of peripherals include General Purpose Input Output (GPIO) for digital input/output, Analog to Digital (ADC) converter for analog inputs, and high-speed serial communications for digital inter-system communication using protocols like I2C and SPI. For digital audio communication, specialized peripherals exist using the I2S or PDM protocols.

Microcontrollers are widely used across all forms of electronics, such as household electronics and mobile devices, telecommunications infrastructure, cars, and industrial systems. In 2017 over 25 billion microcontrollers were shipped and shipments are expected to grow by more than 50% over the next 5 years[82].

Examples of microcontrollers (from ST Microelectronics) that could be used for audio processing are shown in Table 2.2. Similar offerings are available from other manufacturers such as Texas Instruments, Freescale, Atmel, Nordic Semiconductors, NXP.

Name	Architecture	Flash (kB)	RAM (kB)	CPU (MHz)	Price (USD)
STM32F030CC	Cortex-M0	256	32	48	1.0
STM32L476	Cortex-M4	1024	128	80	5.0
STM32F746	Cortex-M7	1024	1024	216	7.5
STM32H743ZI	Cortex-H7	2048	1024	400	9.0

Table 2.2: Examples of available STM32 microcontrollers and their characteristics. Details from ST Microelectronics website.

2.5.1 Machine learning on microcontrollers

For sensor systems, the primary use case for Machine Learning is to train a model on a desktop or cloud system ("off-line" learning), then to deploy the model to the microcontroller to perform inference. Dedicated tools are available for converting models to something that can execute on a microcontroller, usually integrated with established machine learning frameworks.

CMSIS-NN by ARM is a low-level library for ARM Cortex-M microcontrollers implementing basic neural network building blocks, such as 2D convolutions, pooling and Gated Recurrent Units. It uses optimized fixed-point maths and SIMD (Single Instruction Multiple Data) instructions to perform 4x 8-bit operations at a time. This allows it to be up to 4x faster and 5x more energy efficient than floating-point[83].

uTensor[84] by ARM allows running a subset of TensorFlow models on ARM Cortex-M devices, designed for use with the ARM Mbed software platform[85].

TensorFlow Lite for Microcontrollers is an experimental port of TensorFlow[86], announced at TensorFlow Developer Summit in March 2019[87]. Its goal is to be compatible with TensorFlow Lite (for mobile devices et.c.), and to support multiple hardware and software

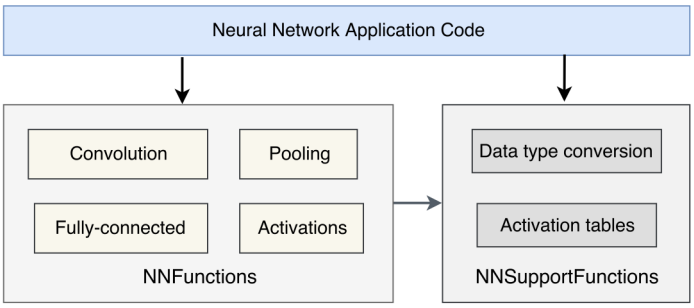


Figure 2.21: Low level functions provided by CMSIS-NN (light gray) for use by higher-level code (light blue)[83]

platforms (not just ARM Cortex). They plan to reuse platform-specific libraries such as CMSIS-NN or uTensor in order to be as efficient as possible.

EdgeML by Microsoft Research India[88] is a research project and open-source code repository which contains novel algorithms developed especially for microcontrollers, such as Bonsai[89], ProtoNN[90] and FastGRNN[61].

Emlearn[91] by the author is a Python library that supports converting a subset of Scikit-Learn[92] and Keras[93] models and run them using C code designed for microcontrollers.

X-CUBE-AI[94] by ST Microelectronics provides official support for performing inference with Neural Networks on their STM32 microcontrollers. It is an add-on to the STM32CubeMX[95] software development kit, and allows loading trained models from various formats, including Keras (Tensorflow[86]), Caffe[96] and PyTorch[97]. In X-CUBE-AI 3.4, all computations are done in single-precision float. Model compression is supported by quantizing model weights by 4x or 8x, but only for fully-connected layers (not convolutional layers)[98]. X-CUBE-AI 3.4 does not use CMSIS-NN.

2.5.2 Hardware accelerators for neural networks

With the increasing interest in deploying neural networks on low-power microcontrollers, dedicated hardware acceleration units are also being developed.

STMicroelectronics (ST) has stated that neural network accelerators will be available for their STM32 family of microcontrollers[99], based on their FD-SOI chip architecture[100].

ARM has announced ARM Helium, an extended instruction set for the Cortex M family of microcontrollers that can be used to speed up neural networks[101].

Kendryte K210 is a microcontroller based on the open RISC-V architecture that includes a convolutional neural network accelerator[102].

GreenWaves GAP8 is a RISC-V chip with 8 cores designed for parallel-processing. They claim a 16x improvement in power efficiency over an ARM Cortex M7 chip[103].

3 Materials

3.1 Dataset

The dataset used for the experiments is Urbansound8K, described in chapter 2.4.1. Figure 3.1 shows example audio spectrograms for each of the 10 classes.

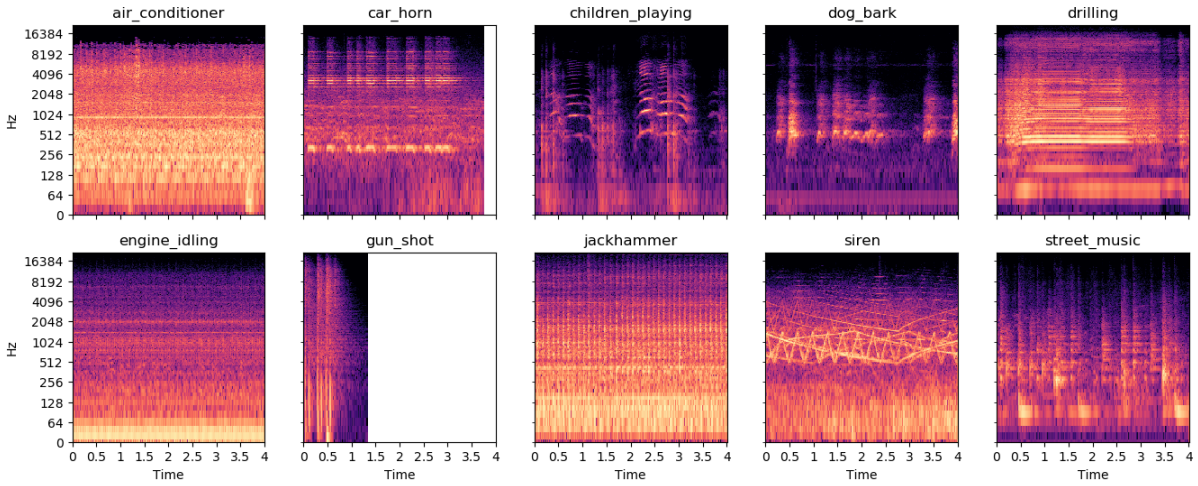


Figure 3.1: Spectrograms of sound clips from Urbansound8k dataset. Audio clips were selected for each class to give clear,representative spectrograms.

The dataset comes prearranged into 10 folds for cross-validation. A single fold may contain multiple clips from the same source file, but the same source file is not used in multiple folds to prevent data leakage.

The target sound is rarely alone in the sound clip and might be in the background, partially obscured by sounds outside the available classes. This makes Urbansound8k a relatively challenging dataset.

3.2 Hardware platform

The microcontroller chosen for this thesis is the STM32L476[104] from STMicroelectronics. This is a mid-range device from ST32L4 series of ultra-low-power microcontroller. It has an ARM Cortex M4F running at 80 MHz, with hardware floating-point unit (FPU) and DSP instructions. It has 1024 kB of program memory (Flash), and 128 kB of RAM.

For audio input, both analog and digital microphones (I2S/PDM) are supported. The microcontroller can also send and receive audio over USB from a host computer. An SD-card interface can be used to record samples to collect a dataset.

To develop for the STM32L476 microcontroller the SensorTile development kit STEVAL-STLKT01V1[105] was selected. The kit consists of a SensorTile module, an expansion board, and a portable docking board (not used).

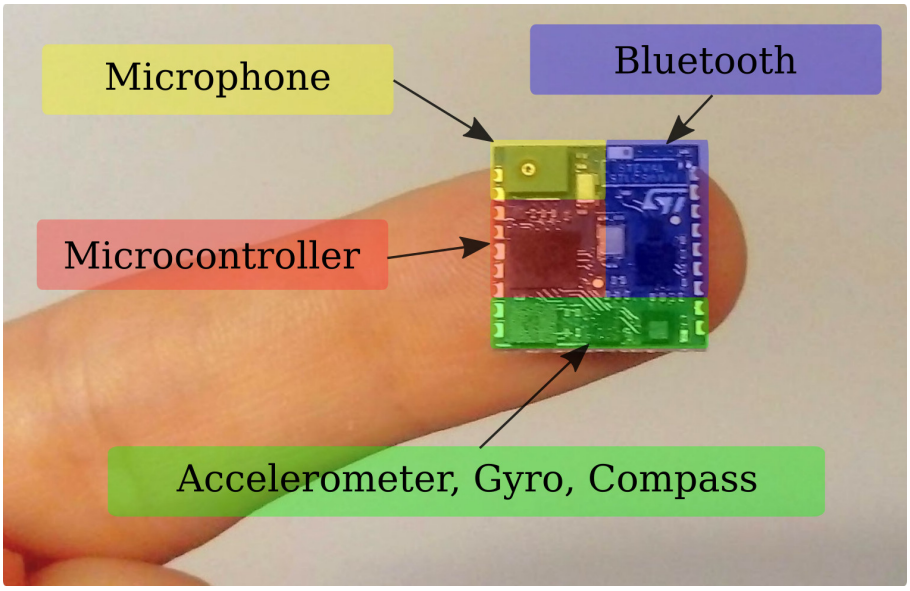


Figure 3.2: SensorTile module with functional blocks indicated. Module size is 13.5x13.5mm

The SensorTile module (see Figure 3.2) contains in addition to the microcontroller: a microphone, Bluetooth radio chip, and an Inertial Measurement Unit (accelerometer+gyroscope+compass). The on-board microphone was used during testing.

An expansion board allows to connect and power the microcontroller over USB. The ST-Link V2 from a Nucleo STM32L476 board is used to program and debug the device. The entire setup can be seen in Figure 3.3.

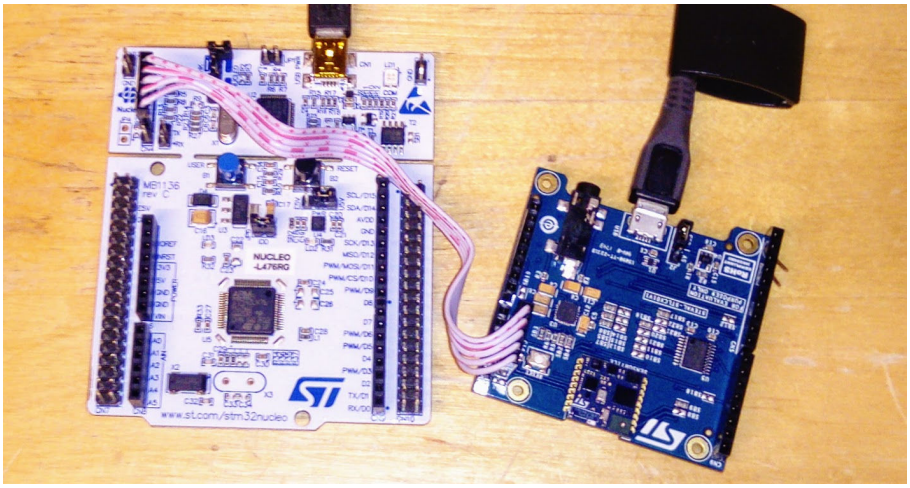


Figure 3.3: Development setup of SensorTile kit

3.3 Software

The STM32L476 microcontroller is supported by the STM32CubeMX[95] development package and the X-CUBE-AI[94] neural network add-on from ST Microelectronics. Version 3.4.0 of X-CUBE-AI was used.

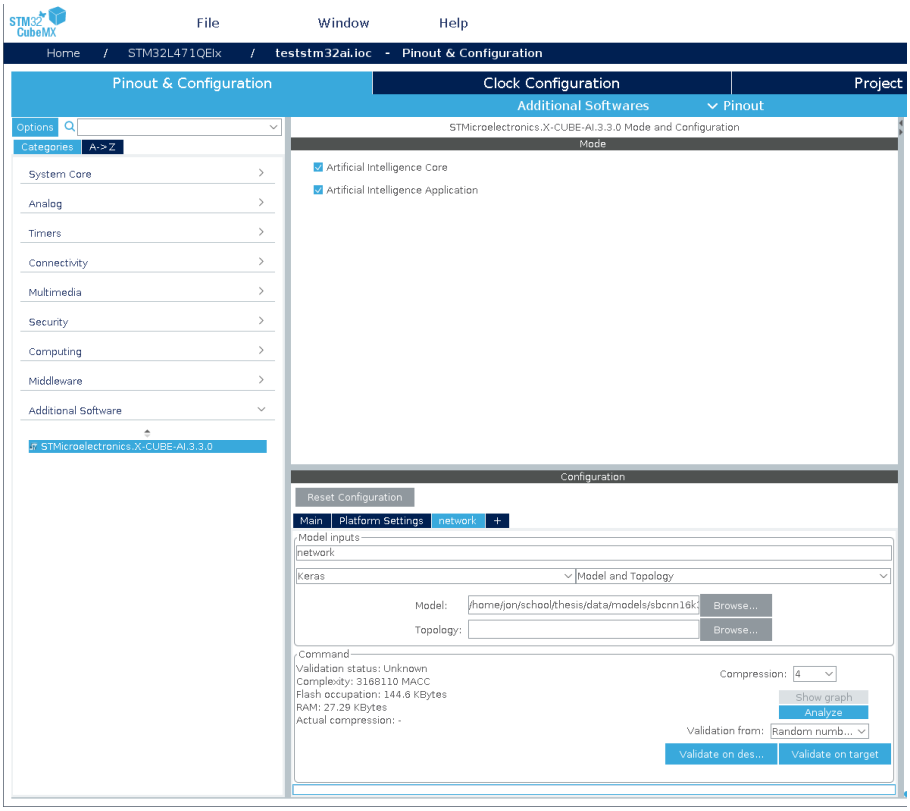


Figure 3.4: STM32CubeMX application with X-CUBE-AI addon after loading a Keras model

A Python command-line script was created to streamline collecting model statistics using X-CUBE-AI, without having to manually use the STM32CubeMX user interface. It is attached in appendix C. This tool provides required Flash storage (in bytes), RAM usage and CPU usage (MACC operations) as JSON, and writes the generated C code to a specified directory.

The training setup is implemented in Python. The machine learning models are implemented in Keras using the Tensorflow backend. To perform feature extraction during training the librosa[106] Python library was used. Numpy and Pandas libraries were used for general numeric computations and data management.

The training setup has automated tests made with the pytest testing framework, and uses Travis CI to execute the tests automatically for each change.

All the code is available at https://github.com/jonnor/ESC-CNN-microcontroller/tree/thesis-submitted. Experiments for the reported results were ran on git commit `b49efa5dde48f9fd72a32eff4c751d9d0c0de712`.

3.4 Models

3.4.1 Model requirements

The candidate models have to fit the constraints of our hardware platform and leave sufficient resources for other parts of an application to run on the device. To do so, a maximum of 50% of the CPU, RAM, and FLASH capacity is allocated to the model.

ST estimates that an ARM Cortex M4F type device uses approximately 9 cycles/MACC[98]. With 80 MHz CPU frequency, this is approximately 9 M MACC/second at 100% CPU utilization. 50% CPU capacity is then estimated as 4.5 M MACC/second. 50% of RAM and FLASH of the microcontroller in use is 64 kB RAM and 512 kB FLASH memory.

For each of these aspects, it is highly beneficial to be well below the hard constraints. If the FLASH and RAM usage can be reduced to half or one-fourth, the cost of the microcontroller is reduced by almost 2/4x. If CPU usage can be reduced to one-tenth, that can reduce power consumption by up to 10 times.

Models from the existing literature (reviewed in chapter 2.4.2) are summarized in Table 3.1 and shown with respect to these model constraints in Figure 3.5. Even the smallest existing models require significantly more than the available resources.

name	Accuracy (%)	MACC / second	Model parameters
Dmix-CNN-mel	82.6	298M	1180k
D-CNN	81.9	458M	33000k
SB-CNN	79.0	25M	432k
LD-CNN	79.0	10M	580k
PiczakCNN	75.0	88M	25534k

Table 3.1: Existing methods and their results on Urbansound8k

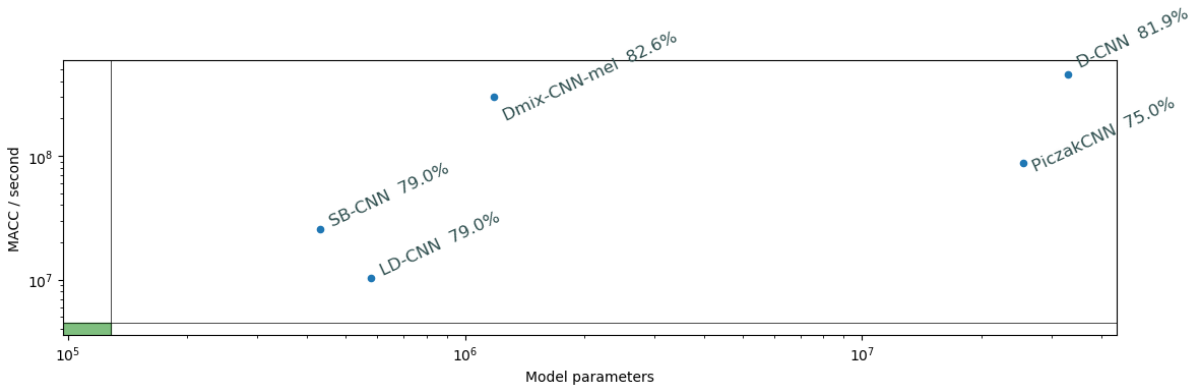


Figure 3.5: Model complexity and accuracy scores of existing CNN models using log-mel features on Urbansound8k dataset. Green area bottom left shows the region which satisfies our model requirements.

3.4.2 Compared models

SB-CNN and LD-CNN are the two best candidates for a baseline model, being the only two that are close to the desired performance characteristic. SB-CNN utilizes a CNN architecture with small uniformly sized kernels (5x5) followed by max pooling, which is very similar to efficient CNN models for image classification. LD-CNN, on the other hand, uses less conventional full-height layers in the start and takes both mel-spectrogram and delta-Mel-spectrogram as inputs. This requires twice as much RAM as a single input and the convolutions in the CNN should be able to learn delta-type features if needed. For these reasons, SB-CNN was used as the base architecture for experiments.

The *Baseline* model has a few modifications from the original SB-CNN model: Max pooling is 3x2 instead of 4x2. Without this change the layers become negative sized due to the reduced input feature size (60 Mel filter bands instead of 128). Batch Normalization was added to each convolutional block. The Keras definition for the Baseline model can be found in appendix A.

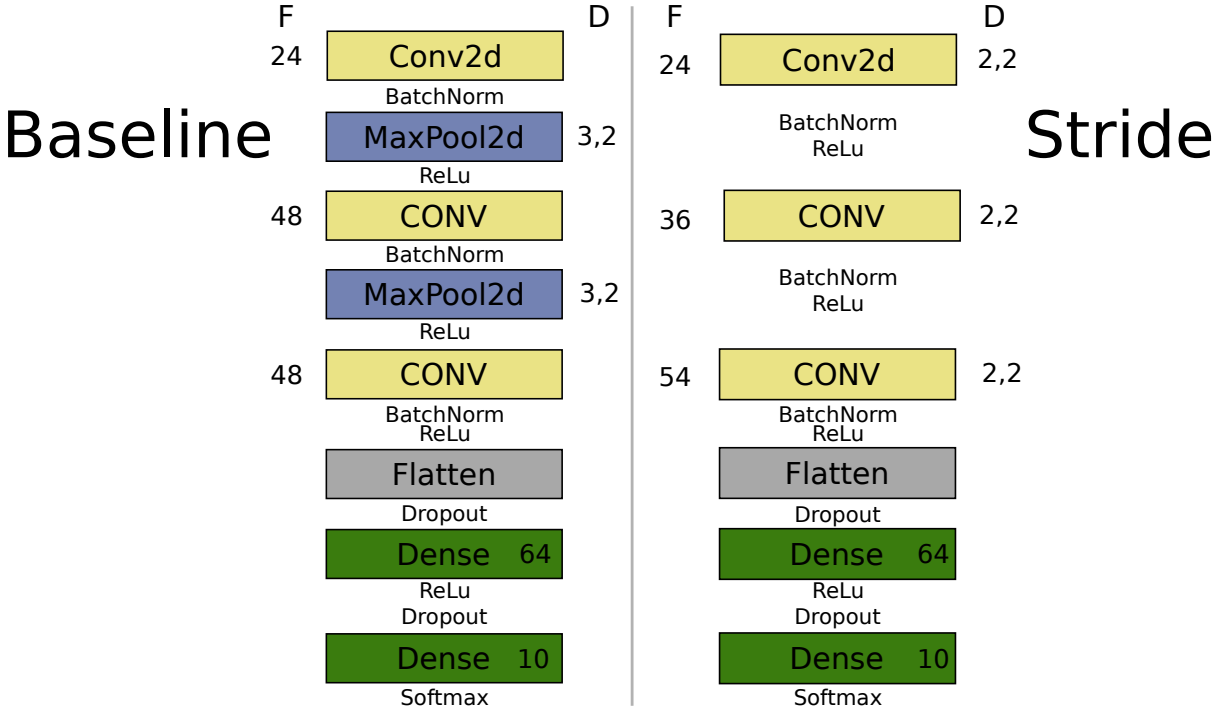


Figure 3.6: Base architecture of the compared models: Baseline and Stride. In Stride the MaxPooling2d operation (blue) has been removed in favor of striding the convolutional blocks. F=Filters D=Downsampling. CONV means a generic convolution block, replaced with the convolution type for different variations

From the baseline architecture, several model variations are created in order to evaluate the effects of using different convolutional blocks and as well as replacing max-pooling with strided convolutions. First, the Baseline was modified with just Depthwise-Separable convolutions (nicknamed Baseline-DS) or striding (nicknamed *Stride*). Since the stride height and width in Keras/Tensorflow must be uniform, 2x2 is used instead of 3x2 from max-pooling. Figure 3.6 illustrates the two architectures.

Three different convolution blocks are tested on top of the Stride model: Depthwise Separable (Stride-DS-*), Bottleneck with Depthwise Separable (Stride-BTLN-DS) and Effnet block (Stride-Effnet). For EffNet, LeakyReLU was with ReLu since LeakyReLU is not supported by X-CUBE-AI, version 3.4. Additionally, a version of strided with Depthwise-Separable convolution with 3x3 kernel size (Stride-DS-3x3) was tested. The Keras definition the strided model(s) can be found in appendix B.

Strided-DS showed promising results in early testing on a single fold. Therefore a few variations with fewer number of filters were also tested, to get an idea for the possible performance/complexity tradeoffs. In addition to the maximum of 24 filters, 20, 16 and 12 filters were tested. For all other Strided models, the number of layers and filters were set as high as possible without violating any of the device constraints.

This results in 10 different models, as summarized in Table 3.2.

Model	Downsample	Convolution	L	F	MACC	RAM	FLASH
Baseline	maxpool 3x2	standard	3	24	10185 K	35 kB	405 kB
Baseline-DS	maxpool 3x2	DS	3	24	1567 K	55 kB	96 kB
Stride	stride 2x2	standard	3	22	2980 K	55 kB	372 kB
Stride-BTLN-DS	stride 2x2	BTLN-DS	3	22	445 K	47 kB	80 kB
Stride-DS-12	stride 2x2	DS	3	12	208 K	27 kB	88 kB
Stride-DS-16	stride 2x2	DS	3	16	291 K	36 kB	118 kB
Stride-DS-20	stride 2x2	DS	3	20	380 K	45 kB	149 kB
Stride-DS-24	stride 2x2	DS	3	24	477 K	54 kB	180 kB
Stride-DS-3x3	stride 2x2	DS	4	24	318 K	54 kB	95 kB
Stride-Effnet	stride 2x2	Effnet	3	22	468 K	47 kB	125 kB

Table 3.2: Parameters of the compared models. L=Number of convolution layers, F=Filters in each convolution layer, DS=Depthwise Separable, BTLN=Bottleneck

Residual connections were not tested, as the networks are relatively shallow. Grouped convolutions were not tested, as they are not supported by X-CUBE-AI version 3.4.

4 | Methods

The method for experimental evaluation of the models follows as closely as possible the established practices for the Urbansound8k dataset.

Figure 4.1 illustrates the overall setup of the classification model.

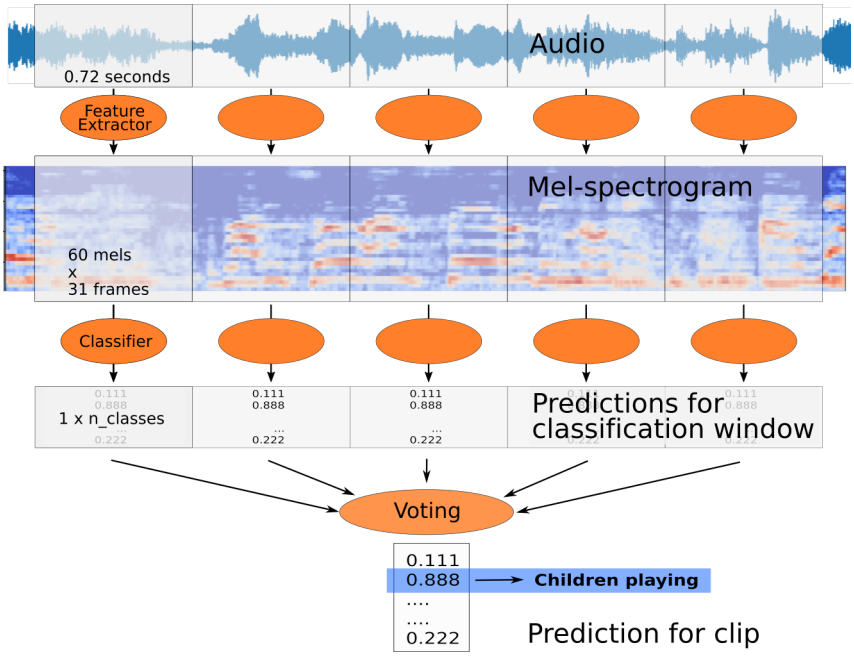


Figure 4.1: Overview of the full model. The classifier runs on individual analysis windows, and predictions for the whole audio clip is done by voting over predictions from all the analysis windows.

4.1 Preprocessing

Mel-spectrograms are used as the input feature. The most compact and most computationally efficient feature-set in use by existing methods was by LD-CNN, which used windows of 31 frames @ 22050 Hz (720 ms) with 60 Mel-filter bands. This has achieved results near the state-of-art, so the same overall settings were used. However, the delta mel-spectrograms were dropped to reduce RAM consumption.

During preprocessing, Data Augmentation is also performed. Time-stretching and Pitch-shifting was done following [74], for a total of 12 variations per sample. The preprocessed mel-spectrograms are stored on disk as Numpy arrays for use during training.

setting	value
Samplerate (Hz)	22050
Melfilter bands	60
FFT length (samples)	1024
FFT hop (samples)	512
Classification window	31
Minibatch size	400
Epochs	100
Training samples/epoch	30000
Validation samples/epoch	5000
Learning rate	0.005
Nesterov momentum	NaN

Table 4.1: Summary of preprocessing and training settings

During training time each window of Mel-spectrogram frames is normalized by subtracting the mean of the window and dividing by the standard deviation.

4.2 Training

The preassigned folds of the Urbansound8k dataset were used. One of the folds was used as a validation set, and one fold was held-out as the test set.

Training was done on individual analysis windows, with each window inheriting the label of the audio clip it belongs to. In each mini-batch, audio clips from the training set are selected randomly. And for each audio clip, a time window is selected from a random position[74]. This effectively implements time-shifting data augmentation.

In order to evaluate the model on the entire audio clip, an additional pass over the validation set is done which combines predictions from multiple time-windows as shown in Figure 4.1.

As the optimizer, Stochastic Gradient Decent (SGD) with Nesterov momentum set to 0.9 is used. The learning rate was set to 0.005 for all models. Each model is trained for up to 100 epochs. A complete summary of experiment settings can be seen in Table 4.1.

Training was performed on an NVidia GTX2060 GPU with 6GB of RAM on a machine with an Intel i5-9400F CPU, 16 GB of RAM and a Kingston A1000 M.2 SSD. However the models can be trained on any device supported by TensorFlow and a minimum of 2GB RAM.

In total 100 training jobs were ran for the experiments, 10 folds for each of 10 tested models. Jobs were processed with 3 jobs in parallel and took approximately 36 hours in total. GPU utilization was only 15%, suggesting that the training process was bottlenecked by the CPU or SSD when preparing the training batches for the GPU.

4.3 Evaluation

Once training is completed, the model epoch with the best performance on the validation set is selected for each of the cross-validation folds. The selected models are then evaluated on the test set in each fold.

In addition to the standard cross-validation for Urbansound8k, the model performance is evaluated on also by separating foreground and background sounds.

The SystemPerformance application skeleton from X-CUBE-AI is used to record the average inference time per sample on the STM32L476 microcontroller. This accounts for potential variations in number of MACC/second for different models, which would be ignored if only relying on the theoretical MACC number.

Finally, the trained models were tested running on the microcontrollers using live audio on the microphone. The on-device test used the example code from the ST FP-SENSING1[107] function pack as a base, with modifications made to send the model predictions out over USB. The example code unfortunately only supports Mel-spectrogram preprocessing with 16 kHz sample-rate, 30 filters and 1024 samples FFT window with 512 hops, using max-normalization for the analysis windows. Therefore a Strided-DS model was trained on fold 1-8 to match these feature settings.

The on-device testing was done ad-hoc with a few samples from Freesound.org, as a sanity-check that the model remained functional when ran on the microcontroller. No systematic measurement of the performance was performed.

5 | Results

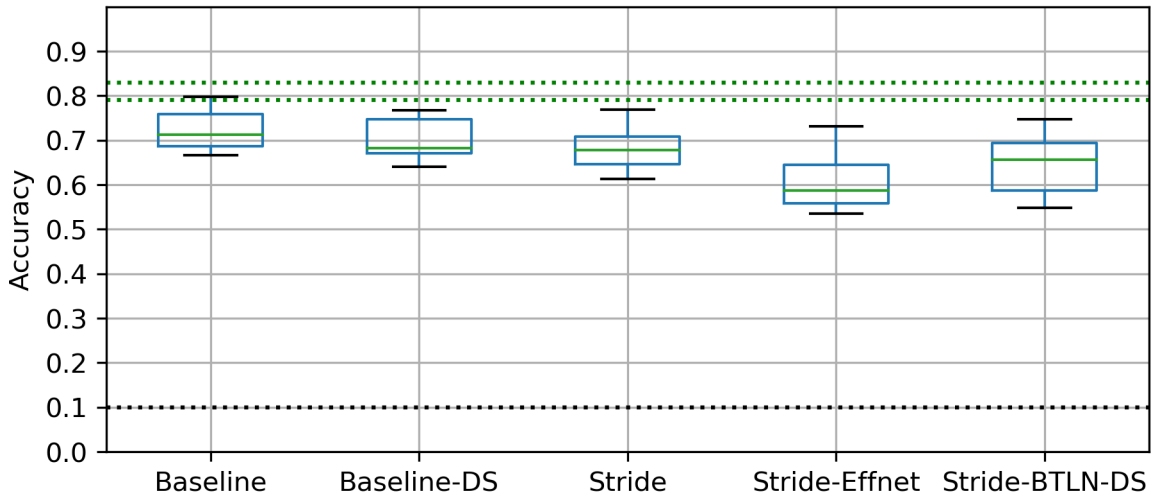


Figure 5.1: Test accuracy of the different models. State-of-the-art averages (SB-CNN/LD-CNN and D-CNN) marked with green dots. No-information rate marked with black dots.

Model	CPU use	Accuracy	FG Accuracy	BG Accuracy
Baseline	971 ms	72.3% ±4.6	78.3% ±7.1	60.5% ±7.7
Baseline-DS	244 ms	70.2% ±4.7	76.1% ±7.5	58.6% ±8.2
Stride	325 ms	68.3% ±5.2	74.1% ±6.6	56.6% ±8.0
Stride-BTLN-DS	71 ms	64.8% ±7.1	69.5% ±8.2	55.3% ±8.9
Stride-DS-12	38 ms	66.0% ±6.0	72.6% ±6.5	53.3% ±9.1
Stride-DS-16	51 ms	67.5% ±5.6	73.3% ±7.7	56.2% ±8.3
Stride-DS-20	66 ms	68.4% ±5.2	75.0% ±7.4	55.2% ±10.0
Stride-DS-24	81 ms	70.9% ±4.3	75.8% ±6.3	61.8% ±6.8
Stride-DS-3x3	59 ms	67.2% ±6.5	73.0% ±7.4	55.8% ±9.1
Stride-Effnet	73 ms	60.7% ±6.6	66.9% ±7.9	48.7% ±8.3

Table 5.1: Results for the compared models. CPU usage as measured on microcontroller. FG=Foreground samples only, BG=Background samples only.

As seen in Table 5.1 and Figure 5.1 the Baseline model gets 72.3% mean accuracy. This is the same level as SB-CNN and PiczakCNN without data-augmentation (73%)[74], but significantly below the 79% of SB-CNN and LD-CNN with data-augmentation. As expected, the Baseline uses more CPU than our requirements with 971 ms classification time per 730 ms analysis window.

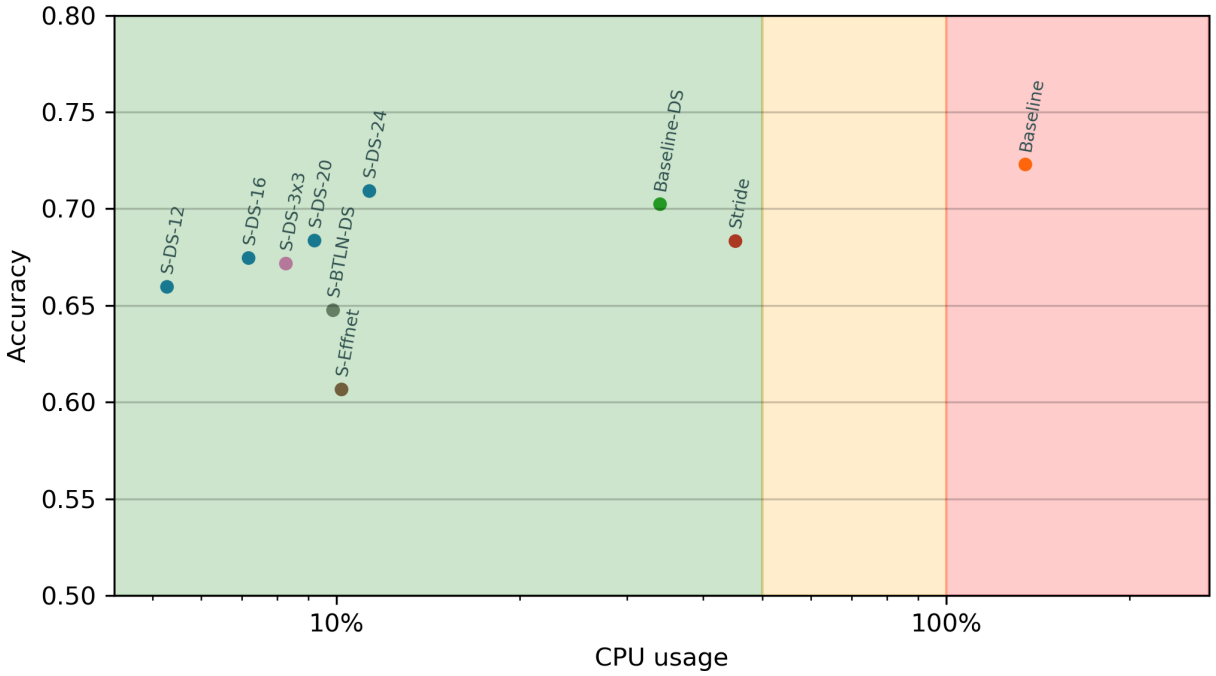


Figure 5.2: Accuracy versus compute of different models. Green section is the area inside our model requirements. Red section is not possible to to classification in real-time. Variations of the same model family have the same color. Strided- has been shortened to S- for readability.

Stride-DS with 70.9% mean accuracy is able to get quite close the baseline performance, despite having (from Table 3.2) $10185/477 = 21x$ fewer multiply-add operations (MACC). The practical efficiency gain in CPU usage is however only $971/81 = 12x$.

Stride-BTLN-DS and Stride-Effnet performed very poorly in comparison. This can be most clearly seen from Figure 5.2. Despite almost the same computational requirements as Stride-DS-24, they had accuracy scores that were 6.1 and 10.2 percentage points lower, respectively.

As seen in Figure 5.3, class accuracies vary widely. The most accurately predicted classes were Gun Shot (96%), Car Horn (86%), Siren (82.7%) and Dog Bark (81.7%). The poorest performance was on the Air Conditioner class (47% accuracy), which was misclassified as almost all the other classes. Drilling (60% accuracy) was often thought to be Jackhammer (20% of the time). Engine Idling (59.9%) was also often thought to be Jackhammer (19% of the time). The remaining classes, Street Music, Children Playing and Jackhammer had around average performance.

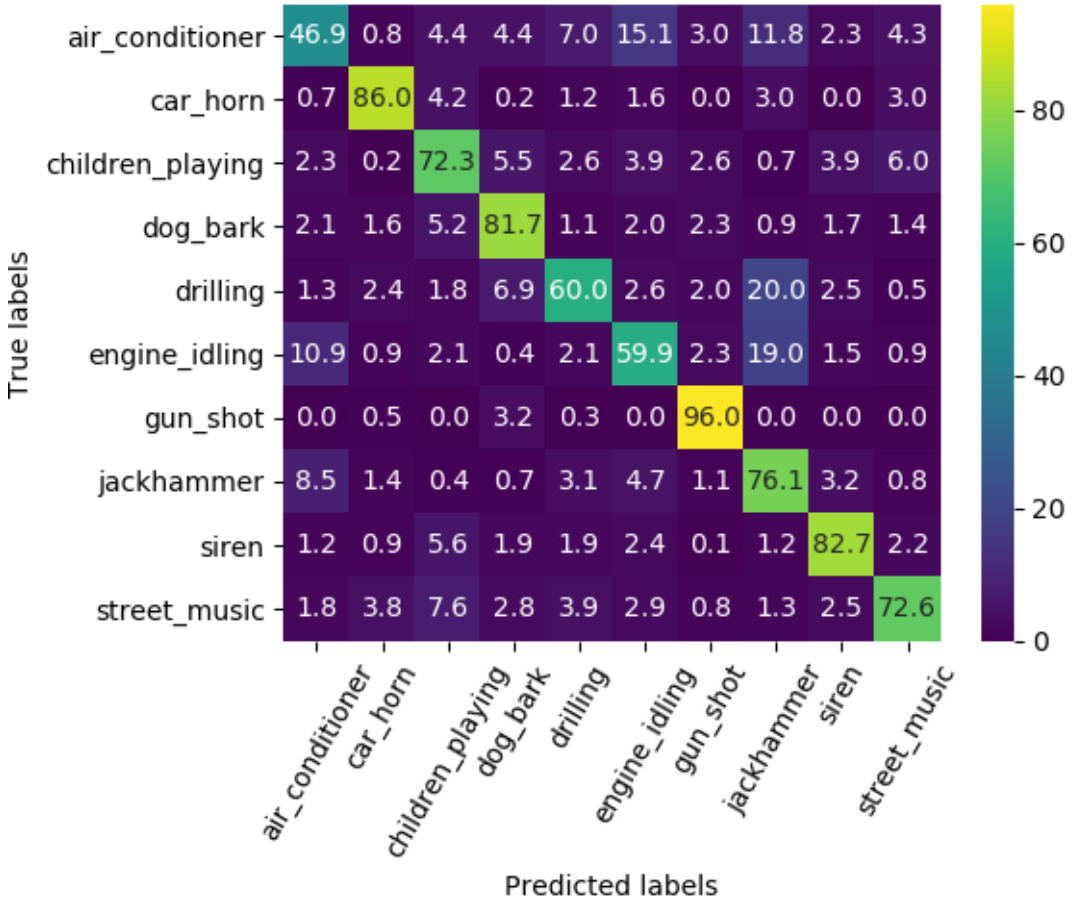


Figure 5.3: Confusion matrix on Urbansound8k. Correct predictions along the diagonal, and misclassifications on the off-diagonal. Normalized to shows percentages.

5.1 On-device testing

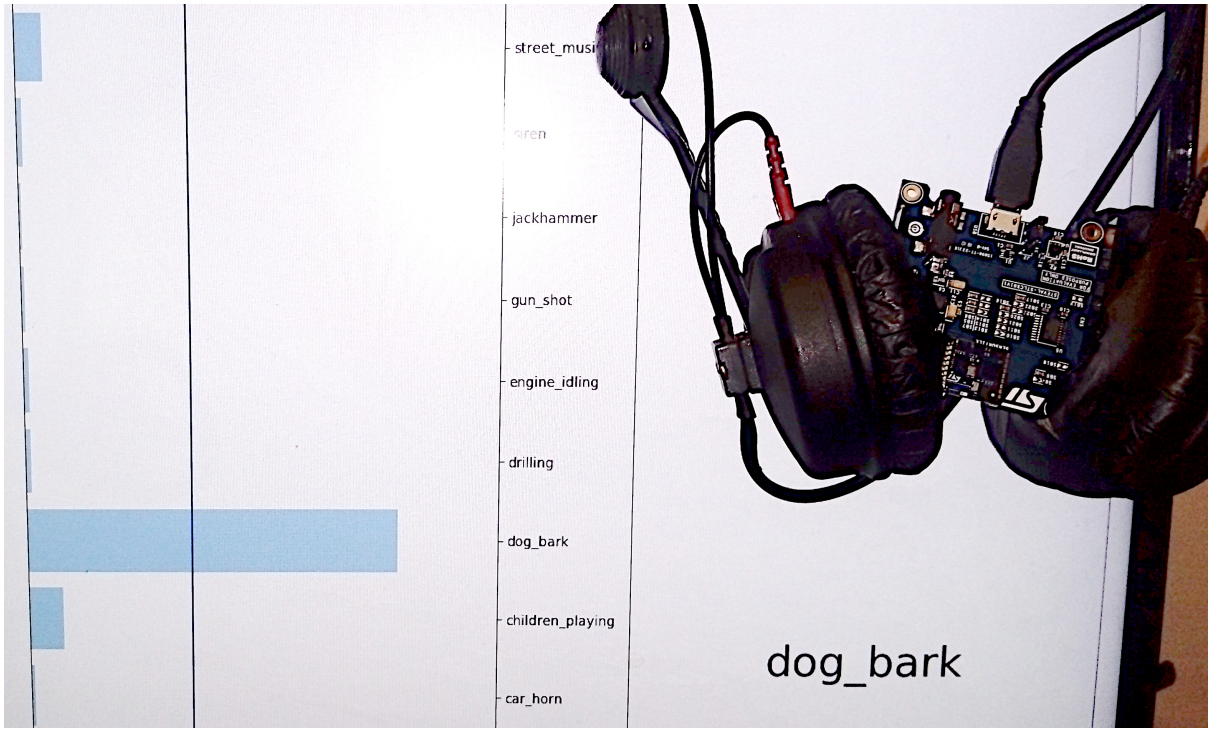


Figure 5.4: Model being tested on device. Sound is played back via headphones and classified on the microcontroller. Predictions are sent to computer and visualized on screen in real-time.

The model used on the device (with 16kHz model with 30 mel filters) scored 72% on the associated validation-set, fold 9. When running on the device, the model execution took 43 ms per analysis window, while preprocessing of the Mel-spectrogram took approximately 60 ms.

Figure 5.4 shows a closeup of the on-device testing scenario. When playing back a few sounds the system was able to correctly classify classes such as “dog barking” most of the time. The classes “jackhammer” and “drilling” were confused several times (in both directions), but these were also often hard to distinguish by ear. The system seemed to struggle with the “children playing” class. When not playing any sound, the GPU fan noise from the nearby machine-learning rig was classified as “air conditioner” - which the author can agree sounded pretty close.

6 | Discussion

6.1 Model comparison

The lower performance of our Baseline relative to SB-CNN/LD-CNN may be a result of the reduced feature representation, or the reduced number of predictions for one clip. Compared to LD-CNN the delta-Mel-spectrogram features are missing, which might have made it easier to learn some patterns - at a cost of twice the RAM and CPU for the first layer. Compared to SB-CNN the analysis window is shorter (720 ms versus 1765 ms), also a 2x reduction in RAM and CPU. Since no overlap is used there are only 6 analysis windows and predictions to be aggregated over a 4 second long clip. In SB-CNN on the other hand, the hop size during validation and test is 1 frame, giving over 100 predictions over a 4 second long clip. This can improve performance, but requires a much higher CPU usage. However, it is also possible that with a more powerful training setup with as transfer learning or stronger data augmentation scheme, that this performance gap could be reduced.

The poorly performing Strided-BTLN-DS and Strided-Effnet both have a bottleneck 1x1 convolution in the start of each block, reducing the number of channels used in the spatial convolution. This hyperparameter was set to a seemingly conservative reduction of 2x (original Effnet used 8x[35], ShuffleNet used 4x[36], albeit on much bigger models). It is possible that this choice of hyperparameter is critical and that other values would have performed better, but this has not been investigated.

Of the models compared it looks like the Strided-DS family of models give the highest accuracy relative to model compute requirements. The largest model, Strided-DS-24, was able to achieve near Baseline performance while utilizing 12x less CPU. The CPU usage of this model is 11%, well within the 50% set as a requirement. The fact that speedup (12x) was not linear with the MACC reduction (21.5x) highlights the importance of measuring execution time on a real device.

The smaller models (with 20,16,12 filters) in the Strided-DS family with less compute requirements had correspondingly lower accuracies. This suggests that a the tradeoff between model complexity and performance can be adjusted. The Strided-DS-3x3 variation with 4 layers with 3x3 convolutions instead was close in performance to the Strided-DS models with 3 layers of 5x5. This could be investigated closer, there may exist variations on this 3x3 model that would perform better than 5x5.

From a one-fold spot check the on-device model trained on 16 kHz sample-rate with 30 Mel filters, looked to perform similarly to those with the full 22 kHz and 60 Mel filters. This

may suggest that perhaps the feature representation (and thus compute requirements) can be reduced even further without much reduction in performance.

6.2 Spectrogram processing time

On-device testing showed that the Mel-feature preprocessing took 60 ms, which is on the same order as the most efficient models during inference (38-81 ms). This means that the CPU bottleneck is not just the model inference time, but that the spectrogram calculation must be also optimized to reach even lower power-consumption. In the FP-SENSING1 example used, the spectrogram computation already uses ARM-specific optimized code from the CMSIS library, albeit with floating-point and not fixed-point.

Taking into account the preprocessing time the Stride-DS-24 model would consume approximately 20% CPU. The microcontroller would be able to sleep for the remaining 80% of the time even when classifications are performed every 720 ms analysis window (real-time).

This is an opportunity for end-to-end models that take raw-audio as input instead of requiring preprocessed spectrograms (ref section 2.4.3), as they might be able to do this more efficiently. When low-power hardware accelerators for Convolutional Neural Networks becomes available, an end-to-end CNN model will become extra interesting, as it would allow also the filterbank processing to be offloaded to the CNN co-processor.

6.3 Practical evaluation

Deployment of noise monitoring systems, and especially systems with noise classification capabilities, are still rare. Of the 5 large-scale research projects mentioned in the introduction, only the SONYC deployment seems to have integrated noise classification capability, using state-of-the-art classification algorithms[6]. They argue that such a system addresses shortcomings of current monitoring and enforcement mechanisms by quantifying the impact of construction-permits, providing historical data to validate noise complaints, and allow better allocation of manual noise inspection crews[6, Ch. 7].

In principle the system proposed here should support the same kind of workflow, and whether the reduced accuracy of 70.9% versus state-of-the-art of 79-83%, impacts the practical use would have to be evaluated on a case-by-case basis.

From a critical perspective, 70.9% on Urbansound8k is likely below human-level performance: While no studies have been done on human-level performance on Urbansound8k directly, it is estimated to be 81.3% for ESC-50 and 95.7% on ESC-10[66, Ch. 3.1], and PiczakCNN which scored 73% on Urbansound8k scored only 62% on ESC-50 and 81% on ESC-10[73].

From an optimistic perspective, the vast majority of cities do not make widespread use of noise monitoring equipment today. So *any* sensor with sound-level logging and even rudimentary classification capabilities would be providing new information that could be used to improve operations. But as with any data-driven system, making use of this information would have to also take into account the limitations of the system.

From table 5.1 it can be seen that the accuracy for foreground sounds is around 5 percentage points better than overall accuracy, reaching above 75%. Background sounds, on the other hand, have a much lower accuracy. The best models score under 62% accuracy, an 8 percentage point drop (or more). This is expected since the signal to noise ratio is lower. If the information of interest is the predominant sound in an area close to the sensor, one could maybe take this into account by only classifying loud (probably closer) sounds, in order to achieve higher precision. Of course, the reduced ability to classify sounds that are far away would require a higher density of sensors, which may be cost-prohibitive.

The accuracy for Urbansound8k classification is based on 4-second intervals. In a noise monitoring situation this granularity of information may not be needed. For example to understand temporal patterns across a day or week, or analyze a noise complaint about a persistent noise, information about the predominant noise source(s) with 15 minute resolution might be enough. For sound-sources with a relatively long duration (much more than 4 seconds), such as children playing, drilling or street music it should be possible to achieve higher accuracy by combining many predictions over time. However, this is unlikely to help for short, intermittent sounds ("events") such as a car horn or a gun-shot. Thankfully performance of the model on these short sounds is considerably better than on the long-term sounds.

Given multiple sensors covering one area, it may also be possible to fuse the individual sensor predictions in order to improve overall predictions.

Finally, for a particular deployment it may be realistic to limit classification to only a few characteristic sound classes. For example in [108], the authors describe doing a 3-way classification on sensor nodes near a rock crushing plant.

7 | Conclusions

Based on the need for wireless sensor systems that can monitor and classify environmental noise, this project has investigated performing noise classification directly on microcontroller-based sensor hardware. This on-sensor classification makes it possible to reduce power-consumption and privacy issues associated with transmitting raw audio or detailed audio fingerprints to a cloud system for classification.

Several Convolutional Neural Networks were designed for the STM32L476 low-power microcontroller using the vendor-provided X-CUBE-AI inference engine. The models were evaluated on the Environmental Sound Classification task using the standard Urbansound8k dataset, and validated briefly by testing real-time classification on a SensorTile device. The best models used Depthwise-Separable convolutions with striding, and were able to reach up to 70.9% mean accuracy while consuming only 20% CPU, and staying within predefined 50% RAM and FLASH storage budgets. To our knowledge, this is the highest reported performance on Urbansound8k on a microcontroller.

This indicates that it is computationally feasible to classify environmental sound on affordable low-power microcontrollers, possibly enabling advanced noise monitoring sensor networks with low costs and high density. Further investigations into the power consumption and practical considerations for applying on-edge Environmental Sound Classification using microcontrollers seems to be worth pursuing.

7.1 Further work

Quantization of the models should further reduce CPU, RAM and FLASH usage. This could be used to fit slightly larger models or to make existing models more efficient. The CMSIS-NN library[83] utilizes 8-bit integer operations and is optimized for the ARM Cortex M4F. However it has also been shown that CNNs can be effectively implemented with as little as 2 bits[109][110][111], and without using any multiplications[112][113].

Utilizing larger amounts of training data might be able to increase performance of the models. Possible techniques for this are transfer learning[114] or applying stronger data augmentation techniques such as Mixup[55] or SpecAugment[115].

In a practical deployment of on-sensor classification, it is still desirable to collect *some* data for evaluation of performance and further training. This could be sampled at random, but an on-sensor implementation of Active Learning[116][117] could be able to make this process more power-efficient.

It is critical for overall power consumption to reduce how often on-sensor classification is performed. This should also benefit from an adaptive sampling strategy. For example to primarily do classification for time-periods that exceed a sound level threshold, or to sample less often when the sound source changes slowly.

Appendix

A | Keras model for Baseline

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten, BatchNormalization
from keras.layers import Convolution2D, MaxPooling2D, SeparableConv2D
from keras.regularizers import l2


def build_model(frames=128, bands=128, channels=1, num_labels=10,
                conv_size=(5,5), conv_block='conv',
                downsample_size=(4,2),
                fully_connected=64,
                n_stages=None, n_blocks_per_stage=None,
                filters=24, kernels_growth=2,
                dropout=0.5,
                use_strides=False):
    """
    Implements SB-CNN model from
    Deep Convolutional Neural Networks and Data Augmentation for Environmental Sound Classification
    Salamon and Bello, 2016.
    https://arxiv.org/pdf/1608.04363.pdf

    Based on https://gist.github.com/jaron/5b17c9f37f351780744aefc74f93d3ae
    but parameters are changed back to those of the original paper authors,
    and added Batch Normalization
    """
    Conv2 = SeparableConv2D if conv_block == 'depthwise_separable' else Convolution2D
    assert conv_block in ('conv', 'depthwise_separable')
    kernel = conv_size
    if use_strides:
        strides = downsample_size
        pool = (1, 1)
    else:
        strides = (1, 1)
        pool = downsample_size

    block1 = [
        Convolution2D(filters, kernel, padding='same', strides=strides,
                      input_shape=(bands, frames, channels)),
        BatchNormalization(),
        MaxPooling2D(pool_size=pool),
        Activation('relu'),
    ]
    block2 = [
        Conv2(filters*kernels_growth, kernel, padding='same', strides=strides),
        BatchNormalization(),
```

```
    MaxPooling2D(pool_size=pool),
    Activation('relu'),
]
block3 = [
    Conv2(filters*kernels_growth, kernel, padding='valid', strides=strides),
    BatchNormalization(),
    Activation('relu'),
]
backend = [
    Flatten(),

    Dropout(dropout),
    Dense(fully_connected, kernel_regularizer=l2(0.001)),
    Activation('relu'),

    Dropout(dropout),
    Dense(num_labels, kernel_regularizer=l2(0.001)),
    Activation('softmax'),
]
layers = block1 + block2 + block3 + backend
model = Sequential(layers)
return model
```

B | Keras model for Strided

```
import numpy as np
from keras.models import Model
from keras.layers import Dense, Dropout, Activation, Flatten, BatchNormalization, Input
from keras.layers import MaxPooling2D, SeparableConv2D, Conv2D, DepthwiseConv2D, ZeroPadding2D


def add_common(x, name):
    x = BatchNormalization(name=name+'_bn')(x)
    x = Activation('relu', name=name+'_relu')(x)
    return x

def conv(x, kernel, filters, downsample, name,
        padding='same'):
    """Regular convolutional block"""
    x = Conv2D(filters, kernel, strides=downsample,
            name=name, padding=padding)(x)
    return add_common(x, name)

def conv_ds(x, kernel, filters, downsample, name,
        padding='same'):
    """Depthwise Separable convolutional block
    (Depthwise->Pointwise)

    MobileNet style"""
    x = SeparableConv2D(filters, kernel, padding=padding,
                strides=downsample, name=name+'_ds')(x)
    return add_common(x, name=name+'_ds')

def conv_bottleneck_ds(x, kernel, filters, downsample, name,
        padding='same', bottleneck=0.5):
    """
    Bottleneck -> Depthwise Separable
    (Pointwise->Depthwise->Pointswise)

    MobileNetV2 style
    """

    if padding == 'valid':
        pad = ((0, kernel[0]//2), (0, kernel[0]//2))
        x = ZeroPadding2D(padding=pad, name=name+'pad')(x)

    x = Conv2D(int(filters*bottleneck), (1,1),
                padding='same', strides=downsample,
                name=name+'_pw')(x)
    add_common(x, name+'_pw')
```

```
    x = SeparableConv2D(filters, kernel,
                padding=padding, strides=(1,1),
                name=name+'_ds')(x)
    return add_common(x, name+'_ds')


def conv_effnet(x, kernel, filters, downsample, name,
        bottleneck=0.5, strides=(1,1), padding='same', bias=False):
    """Pointwise -> Spatially Separable conv&pooling
    Effnet style"""

    assert downsample[0] == downsample[1]
    downsample = downsample[0]
    assert kernel[0] == kernel[1]
    kernel = kernel[0]

    ch_in = int(filters*bottleneck)
    ch_out = filters

    if padding == 'valid':
        pad = ((0, kernel//2), (0, kernel//2))
        x = ZeroPadding2D(padding=pad, name=name+'pad')(x)

    x = Conv2D(ch_in, (1, 1), strides=downsample,
            padding=padding, use_bias=bias, name=name+'pw')(x)
    x = add_common(x, name=name+'pw')

    x = DepthwiseConv2D((1, kernel),
            padding=padding, use_bias=bias, name=name+'dwv')(x)
    x = add_common(x, name=name+'dwv')

    x = DepthwiseConv2D((kernel, 1), padding='same',
            use_bias=bias, name=name+'dwh')(x)
    x = add_common(x, name=name+'dwh')

    x = Conv2D(ch_out, (1, 1), padding=padding, use_bias=bias, name=name+'rh')(x)
    return add_common(x, name=name+'rh')


block_types = {
    'conv': conv,
    'depthwise_separable': conv_ds,
    'bottleneck_ds': conv_bottleneck_ds,
    'effnet': conv_effnet,
}

def backend_dense1(x, n_classes, fc=64, regularization=0.001, dropout=0.5):
    from keras.regularizers import l2
    """
    SB-CNN style classification backend
    """

    x = Flatten()(x)
    x = Dropout(dropout)(x)
    x = Dense(fc, kernel_regularizer=l2(regularization))(x)
    x = Activation('relu')(x)

    x = Dropout(dropout)(x)
```

```
    x = Dense(n_classes, kernel_regularizer=l2(regularization))(x)
    x = Activation('softmax')(x)
    return x


def build_model(frames=128, bands=128, channels=1, n_classes=10,
                conv_size=(5,5),
                conv_block='conv',
                downsample_size=(2,2),
                n_stages=3, n_blocks_per_stage=1,
                filters=24, kernels_growth=1.5,
                fully_connected=64,
                dropout=0.5, l2=0.001):
    """

    """
    input = Input(shape=(bands, frames, channels))
    x = input

    block_no = 0
    for stage_no in range(0, n_stages):
        for b_no in range(0, n_blocks_per_stage):
            # last padding == valid
            padding = 'valid' if block_no == (n_stages*n_blocks_per_stage)-1 else 'same'
            # downsample only one per stage
            downsample = downsample_size if b_no == 0 else (1, 1)
            # first convolution is standard
            conv_func = conv if block_no == 0 else block_types.get(conv_block)
            name = "conv{}".format(block_no)

            x = conv_func(x, conv_size, int(filters), downsample,
                          name=name, padding=padding)
            block_no += 1
        filters = filters * kernels_growth

    x = backend_dense1(x, n_classes, fully_connected, regularization=l2)
    model = Model(input, x)
    return model
```

C | Script for converting models using X-CUBE-AI

```
"""
Convert a Keras/Lasagne/Caffe model to C for STM32 microcontrollers using ST X-CUBE-AI

Wrapper around the 'generatecode' tool used in STM32CubeMX from the X-CUBE-AI addon
"""

import pathlib
import json
import subprocess
import argparse
import os.path
import re
import platform

model_options = {
    'keras': 1,
    'lasagne': 2,
    'caffee': 3,
    'convnetjs': 4,
}

def generate_config(model_path, out_path, name='network', model_type='keras', compression=None):

    data = {
        "name": name,
        "toolbox": model_options[model_type],
        "models": {
            "1": [ model_path , ""],
            "2": [ model_path , ""],
            "3": [ model_path , ""],
            "4": [ model_path ],
        },
        "compression": compression,
        "pinnr_path": out_path,
        "src_path": out_path,
        "inc_path": out_path,
        "plot_file": os.path.join(out_path, "network.png"),
    }
    return json.dumps(data)

def parse_with_unit(s):
    number, unit = s.split()
    number = float(number)
```

```
    multipliers = {
        'KBytes': 1e3,
        'MBytes': 1e6,
    }
    mul = multipliers[unit]
    return number * mul

def extract_stats(output):
    regex = r"  ([^:]*):(.*)"

    out = {}
    matches = re.finditer(regex, output.decode('utf-8'), re.MULTILINE)

    for i, match in enumerate(matches, start=1):
        key, value = match.groups()
        key = key.strip()
        value = value.strip()

        if key == 'MACC / frame':
            out['maccs_frame'] = int(value)
            pass
        elif key == 'RAM size':
            ram, min = value.split('(Minimum:')
            out['ram_usage_max'] = parse_with_unit(ram)
            out['ram_usage_min'] = parse_with_unit(min.rstrip(')'))
            pass
        elif key == 'ROM size':
            out['flash_usage'] = parse_with_unit(value)
            pass

    return out


def test_ram_use():
    examples = [
    ("""
    AI_ARRAY_OBJ_DECLARE(
      input_1_output_array, AI_DATA_FORMAT_FLOAT,
      NULL, NULL, 1860,
      AI_STATIC)
    AI_ARRAY_OBJ_DECLARE(
      conv2d_1_output_array, AI_DATA_FORMAT_FLOAT,
      NULL, NULL, 29760,
      AI_STATIC)
    """,
    { 'input_1_output_array': 1860, 'conv2d_1_output_array': 29760 }),

    ]

    for input, expected in examples:
        out = extract_ram_use(input)
        assert out == expected, out


def extract_ram_use(str):
    regex = r"AI_ARRAY_OBJ_DECLARE\(([^)]*)\)"
    matches = re.finditer(regex, str, re.MULTILINE)
```

```
    out = {}
    for i, match in enumerate(matches):
        (items, ) = match.groups()
        items = [ i.strip() for i in items.split(',') ]
        name, format, _, _, size, modifiers = items
        out[name] = int(size)

    return out


def generatecode(model_path, out_path, name, model_type, compression):
    # Path to CLI tool
    home = str(pathlib.Path.home())
    version = os.environ.get('XCUBEAI_VERSION', '3.4.0')
    platform_name = platform.system().lower()
    if platform_name == 'darwin':
        platform_name = 'mac'
    p = 'STM32Cube/Repository/Packs/STMicroelectronics/X-CUBE-AI/{version}/Utilities/{os}/generateco
    default_path = os.path.join(home, p)
    cmd_path = os.environ.get('XCUBEAI_GENERATECODE', default_path)

    # Create output dirs if needed
    if not os.path.exists(out_path):
        os.makedirs(out_path)

    # Generate .ai config file
    config = generate_config(model_path, out_path, name=name,
                             model_type=model_type, compression=compression)
    config_path = os.path.join(out_path, 'config.ai')
    with open(config_path, 'w') as f:
        f.write(config)

    # Run generatecode
    args = [
        cmd_path,
        '--auto',
        '-c', config_path,
    ]
    stdout = subprocess.check_output(args, stderr=subprocess.STDOUT)

    # Parse MACCs / params from stdout
    stats = extract_stats(stdout)
    assert len(stats.keys()), 'No model output. Stdout: {}'.format(stdout)

    with open(os.path.join(out_path, 'network.c'), 'r') as f:
        network_c = f.read()
        ram = extract_ram_use(network_c)
        stats['arrays'] = ram

    return stats

def parse():

    parser = argparse.ArgumentParser(description='Process some integers.')
    a = parser.add_argument

    supported_types = '|'.join(model_options.keys())
```

```
    a('model', metavar='PATH', type=str,
            help='The model to convert')
    a('out', metavar='DIR', type=str,
            help='Where to write generated output')

    a('--type', default='keras',
        help='Type of model. {}'.format(supported_types))
    a('--name', default='network',
        help='Name of the generated network')
    a('--compression', default=None, type=int,
        help='Compression setting to use. Valid: 4|8')


    args = parser.parse_args()
    return args

def main():
    args = parse()

    test_ram_use()

    stats = generatecode(args.model, args.out,
                         name=args.name,
                         model_type=args.type,
                         compression=args.compression)
    print('Wrote model to', args.out)
    print('Model status: ', json.dumps(stats))

if __name__ == '__main__':
    main()
```

References

[1] W. R. O. for Europe, *Burden of disease from environmental noise - quantification of healthy life years lost in europe*. 2018.

[2] W. Babisch, "The noise/stress concept, risk assessment and research needs," *Noise and Health*, vol. 4, no. 16, pp. 1–11, 2002.

[3] "EU directive 2002/49/ec." [Online]. Available: http://ec.europa.eu/environment/noise/directive_en.htm.

[4] "Dublin city nosie website." [Online]. Available: http://www.dublincitynoise.com.

[5] "Sound of new york (sonyc) homepage." [Online]. Available: http://wp.nyu.edu/sonyc.

[6] J. P. Bello *et al.*, "SONYC: A system for monitoring, analyzing, and mitigating urban noise pollution," *Communications of the ACM*, vol. 62, no. 2, pp. 68–77, Feb. 2019.

[7] J. C. Farrés, "Barcelona noise monitoring network," in *Proceedings of the euronoise*, 2015, pp. 218–220.

[8] J. C. Farrés and J. C. Novas, "Issues and challenges to improve the barcelona noise monitoring network," 2018.

[9] "CENSE project homepage." [Online]. Available: http://cense.ifsttar.fr/en/.

[10] J. Ardouin *et al.*, "An innovative low cost sensor for urban sound monitoring," in *INTER-noise and noise-con congress and conference proceedings*, 2018, vol. 258, pp. 2226–2237.

[11] "Citizen scientists can help mitigate noise pollution," 2019. [Online]. Available: https://steinhardt.nyu.edu/site/ataglance/2019/02/citizen-scientists-can-help-mitigate-noise-pollution.html.

[12] "IEC 61672-1 sound level meters, part 1: Specifications." 2013.

[13] A. S. of America, "ANSI s1.4-2014, part 1: Specifications for sound level meters." 2014.

[14] F. Gontier, M. Lagrange, P. Aumond, A. Can, and C. Lavandier, "An efficient audio coding scheme for quantitative and qualitative large scale acoustic monitoring using the sensor grid approach," *Sensors*, vol. 17, no. 12, 2017.

[15] X. Rong, "Word2vec parameter learning explained," *arXiv preprint arXiv:1411.2738*, 2014.

[16] S. Hershey *et al.*, "CNN architectures for large-scale audio classification," in *International conference on acoustics, speech and signal processing (icassp)*, 2017.

[17] J. F. Gemmeke *et al.*, "Audio set: An ontology and human-labeled dataset for audio events," in *Proc. IEEE icassp 2017*, 2017.

[18] R. Arandjelovic and A. Zisserman, "Look, listen and learn," in *Proceedings of the ieee international conference on computer vision*, 2017, pp. 609–617.

[19] S. Kumari, D. Roy, M. Cartwright, J. P. Bello, and A. Arora, "EdgeL3: Compressing l3-net for mote-scale urban noise monitoring."

[20] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain." *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[21] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314, 1989.

[22] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (icml-10)*, 2010, pp. 807–814.

[23] D. E. Rumelhart, G. E. Hinton, R. J. Williams, and others, "Learning representations by back-propagating errors," *Cognitive modeling*, vol. 5, no. 3, p. 1, 1988.

[24] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient backprop," in *Neural networks: Tricks of the trade: Second edition*, G. Montavon, G. B. Orr, and K.-R. Müller, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 9–48.

[25] A. Choromanska, M. Henaff, M. Mathieu, G. B. Arous, and Y. LeCun, "The loss surfaces of multilayer networks," in *Artificial intelligence and statistics*, 2015, pp. 192–204.

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

[27] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[28] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 ieee conference on computer vision and pattern recognition*, 2009, pp. 248–255.

[29] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, "Striving for simplicity: The all convolutional net," *arXiv preprint arXiv:1412.6806*, 2014.

[30] 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*, 2016, pp. 770–778.

[31] Y. Uchida, "Why mobilenet and its variants (e.g. ShuffleNet) are fast." [Online]. Available: https://medium.com/@yu4u/why-mobilenet-and-its-variants-e-g-shufflenet-are-fast-1c7048b9618d.

[32] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size," *arXiv:1602.07360*, 2016.

[33] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[34] A. G. Howard *et al.*, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[35] I. Freeman, L. Roese-Koerner, and A. Kummert, "Effnet: An efficient structure for convolutional neural networks," in *2018 25th ieee international conference on image processing (icip)*, 2018, pp. 6–10.

[36] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," in *Proceedings of the ieee conference on computer vision and pattern recognition*, 2018, pp. 6848–6856.

[37] A. Gholami *et al.*, "SqueezeNext: Hardware-aware neural network design," *CoRR*, vol. abs/1803.10615, 2018.

[38] I. E. C. T. 100, "IEC 60908:1999 audio recording - compact disc digital audio system." 1999.

[39] Microsoft, "WAVE specifications, version 1.0, 1991-08." 1991.

[40] X. Foundation, "FLAC project homepage (free lossless audio codec)." [Online]. Available: https://xiph.org/flac/.

[41] I. E. C. J. 1/SC 29, "ISO/iec 11172-3:1993 coding of moving pictures and associated audio for digital storage media at up to about 1,5 mbit/s – part 3: Audio.".

[42] J. O. Smith, *Spectral audio signal processing.* `//ccrma.stanford.edu/~jos/sasp/`.

[43] J. Allen, "Short term spectral analysis, synthesis, and modification by discrete fourier transform," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 25, no. 3, pp. 235–238, 1977.

[44] D. Griffin and J. Lim, "Signal estimation from modified short-time fourier transform," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 32, no. 2, pp. 236–243, 1984.

[45] S. Ö. Arık, H. Jun, and G. Diamos, "Fast spectrogram inversion using multi-head convolutional neural networks," *IEEE Signal Processing Letters*, vol. 26, no. 1, pp. 94–98, 2019.

[46] M. Anusuya and S. Katti, "Front end analysis of speech recognition: A review," *International Journal of Speech Technology*, vol. 14, no. 2, pp. 99–145, 2011.

[47] M. Huzaifah, "Comparison of time-frequency representations for environmental sound classification using convolutional neural networks," *arXiv preprint arXiv:1706.07156*, 2017.

[48] D. Stowell and M. D. Plumbley, "Automatic large-scale classification of bird sounds is strongly improved by unsupervised feature learning," *PeerJ*, vol. 2, p. e488, Jul. 2014.

[49] S. W. Smith and others, "The scientist and engineer's guide to digital signal processing," 1997.

[50] T. Virtanen, M. Plumbley, and D. Ellis, *Computational analysis of sound scenes and events*. 2017, pp. 1–422.

[51] B. Babenko, "Multiple instance learning: Algorithms and applications," *View Article PubMed/NCBI Google Scholar*, pp. 1–19, 2008.

[52] A. Kumar and B. Raj, "Audio event detection using weakly labeled data," in *Proceedings of the 24th acm international conference on multimedia*, 2016, pp. 1038–1047.

[53] B. McFee, J. Salamon, and J. P. Bello, "Adaptive pooling operators for weakly labeled sound event detection," *IEEE/ACM Transactions on Audio, Speech and Language Processing (TASLP)*, vol. 26, no. 11, pp. 2180–2193, 2018.

[54] V. Morfi and D. Stowell, "Data-efficient weakly supervised learning for low-resource audio event detection using deep learning," *arXiv preprint arXiv:1807.06972*, 2018.

[55] H. Zhang, M. Cisse, Y. N. Dauphin, and D. Lopez-Paz, "Mixup: Beyond empirical risk minimization," *arXiv preprint arXiv:1710.09412*, 2017.

[56] 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)*, 2018, pp. 356–367.

[57] J. J. Huang and J. J. A. Leanos, "AclNet: Efficient end-to-end audio classification cnn," *arXiv preprint arXiv:1811.06669*, 2018.

[58] K. Xu *et al.*, "Mixup-based acoustic scene classification using multi-channel convolutional neural network," in *Advances in multimedia information processing – pcm 2018*, 2018, pp. 14–23.

[59] T. Sainath and C. Parada, "Convolutional neural networks for small-footprint keyword spotting," 2015.

[60] Y. Zhang, N. Suda, L. Lai, and V. Chandra, "Hello edge: Keyword spotting on microcontrollers," *arXiv preprint arXiv:1711.07128*, 2017.

[61] A. Kusupati, M. Singh, K. Bhatia, A. Kumar, P. Jain, and M. Varma, "FastGRNN: A fast, accurate, stable and tiny kilobyte sized gated recurrent neural network," 2018.

[62] S. Chachada and C.-C. J. Kuo, "Environmental sound recognition: A survey," *APSIPA Transactions on Signal and Information Processing*, vol. 3, p. e14, 2014.

[63] J. Salamon, C. Jacoby, and J. P. Bello, "A dataset and taxonomy for urban sound research," in *22nd acm international conference on multimedia (acm-mm'14)*, 2014, pp. 1041–1044.

[64] F. Font, G. Roma, and X. Serra, "Freesound technical demo." ACM; ACM, Barcelona, Spain, pp. 411–412, 2013.

[65] F. Medhat, D. Chesmore, and J. Robinson, "Masked conditional neural networks for environmental sound classification," in *International conference on innovative techniques and applications of artificial intelligence*, 2017, pp. 21–33.

[66] K. J. Piczak, "ESC: Dataset for Environmental Sound Classification," in *Proceedings of the 23rd Annual ACM Conference on Multimedia*, 2015, pp. 1015–1018.

[67] K. J. Piczak, "Karoldvl/esc-50 on github." [Online]. Available: https://github.com/karoldvl/ESC-50.

[68] "DCASE2019 task 5, urban sound tagging," 2019. [Online]. Available: http://dcase.community/challenge2019/task-urban-sound-tagging.

[69] M. Cartwright *et al.*, "SONYC Urban Sound Tagging (SONYC-UST): a multilabel dataset from an urban acoustic sensor network." Mar-2019.

[70] "DCASE2017 task 1, acoustic scene classification," 2017. [Online]. Available: http://www.cs.tut.fi/sgn/arg/dcase2017/challenge/task-acoustic-scene-classification.

[71] "DCASE2018 task 2, general purpose audio tagging using audioset ontology," 2018. [Online]. Available: http://dcase.community/challenge2018/task-general-purpose-audio-tagging.

[72] "DCASE2017 task 4, large-scale weakly supervised sound event detection for smart cars," 2017. [Online]. Available: http://www.cs.tut.fi/sgn/arg/dcase2017/challenge/task-large-scale-sound-event-detection.

[73] K. J. Piczak, "Environmental sound classification with convolutional neural networks," in *2015 ieee 25th international workshop on machine learning for signal processing (mlsp)*, 2015, pp. 1–6.

[74] J. Salamon and J. P. Bello, "Deep convolutional neural networks and data augmentation for environmental sound classification," *CoRR*, vol. abs/1608.04363, 2016.

[75] X. Zhang, Y. Zou, and W. Shi, "Dilated convolution neural network with leakyrelu for environmental sound classification," 2017, pp. 1–5.

[76] Y. Tokozume and T. Harada, "Learning environmental sounds with end-to-end convolutional neural network," 2017, pp. 2721–2725.

[77] Y. Tokozume, Y. Ushiku, and T. Harada, "Learning from between-class examples for deep sound recognition," *arXiv preprint arXiv:1711.10282*, 2017.

[78] W. Dai, C. Dai, S. Qu, J. Li, and S. Das, "Very deep convolutional neural networks for raw waveforms," in *2017 ieee international conference on acoustics, speech and signal processing (icassp)*, 2017, pp. 421–425.

[79] X. Jin *et al.*, "WSNet: Compact and efficient networks through weight sampling," in *Proceedings of the 35th international conference on machine learning*, 2018, vol. 80, pp. 2352–2361.

[80] X. Zhang, Y. Zou, and W. Wang, "LD-cnn: A lightweight dilated convolutional neural network for environmental sound classification," in *2018 24th international conference on pattern recognition (icpr)*, 2018, pp. 373–378.

[81] J. Amoh and K. Odame, "An optimized recurrent unit for ultra-low-power keyword spotting," *arXiv preprint arXiv:1902.05026*, 2019.

[82] I. Insights, "MCUs sales to reach record-high annual revenues through 2022," Nov-2018. [Online]. Available: http://www.icinsights.com/news/bulletins/MCUs-Sales-To-Reach-RecordHigh-Annual-Revenues-Through-2022/.

[83] L. Lai, N. Suda, and V. Chandra, "CMSIS-NN: efficient neural network kernels for arm cortex-m cpus," *CoRR*, vol. abs/1801.06601, 2018.

[84] N. Tan, "UTensor: AI inference library based on mbed and tensorflow.".

[85] "ARM mbed project homepage." [Online]. Available: https://www.mbed.com.

[86] "TensorFlow: Large-scale machine learning on heterogeneous systems." 2015.

[87] P. Warden, "Launching tensorflow lite for microcontrollers," Mar-2019. [Online]. Available: https://petewarden.com/2019/03/07/launching-tensorflow-lite-for-microcontrollers/.

[88] "Edge machine learning by microsoft on github.".

[89] A. Kumar, S. Goyal, and M. Varma, "Resource-efficient machine learning in 2 kb ram for the internet of things," in *Proceedings of the 34th international conference on machine learning-volume 70*, 2017, pp. 1935–1944.

[90] C. Gupta *et al.*, "Protomm: Compressed and accurate knn for resource-scarce devices," in *Proceedings of the 34th international conference on machine learning-volume 70*, 2017, pp. 1331–1340.

[91] J. Nordby, "emlearn: Machine Learning inference engine for Microcontrollers and Embedded Devices." Mar-2019.

[92] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[93] F. Chollet and others, "Keras." https://keras.io, 2015.

[94] "X-cube-ai: AI expansion pack for stm32cubemx," 2019. [Online]. Available: https://www.st.com/en/embedded-software/x-cube-ai.html.

[95] "STM32CubeMX application homepage." [Online]. Available: https://www.st.com/en/development-tools/stm32cubemx.html.

[96] Y. Jia *et al.*, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[97] A. Paszke *et al.*, "Automatic differentiation in pytorch," in *NIPS-w*, 2017.

[98] STMicroelectronics, "UM2526: Getting started with x-cube-ai expansion package for artificial intelligence (ai).".

[99] S. Microelectronics, "Demo ai slides @ stmicroelectronics nv 2018 capital markets day," 15-May-2018. [Online]. Available: http://investors.st.com/events/event-details/stmicroelectronics-nv-2018-capital-markets-day.

[100] G. Desoli *et al.*, "The orlando project: A 28 nm fd-soi low memory embedded neural network asic," in *Advanced concepts for intelligent vision systems*, 2016, pp. 217–227.

[101] "Next-generation armv8.1-m architecture: Delivering enhanced machine learning and signal processing for the smallest embedded devices," 14-Feb-2019. [Online]. Available: ht tps://www.arm.com/company/news/2019/02/next-generation-armv8-1-m-architecture.

[102] Kendryte, "K210 datasheet [english]." [Online]. Available: https://s3.cn-north-1.amazonaws.com.cn/dl.kendryte.com/documents/kendryte_datasheet_20181011163248_en.pdf.

[103] G. Technologies, "GAP8 performance versus arm m7 on embedded cnns." [Online]. Available: https://greenwaves-technologies.com/gap8-versus-arm-m7-embedded-cnns.

[104] STMicroelectronics, "DS10198: STM32L476xx datasheet.".

[105] STMicroelectronics, "STEVAL-stlkt01v1 product information.".

[106] B. McFee *et al.*, "Librosa/librosa: 0.6.3." Feb-2019.

[107] STMicroelectronics, "FP-ai-sensing1 function pack.".

[108] P. Maijala, Z. Shuyang, T. Heittola, and T. Virtanen, "Environmental noise monitoring using source classification in sensors," *Applied Acoustics*, vol. 129, pp. 258–267, 2018.

[109] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "YodaNN: An ultra-low power convolutional neural network accelerator based on binary weights," in *2016 ieee computer society annual symposium on vlsi (isvlsi)*, 2016, pp. 236–241.

[110] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional neural networks using logarithmic data representation," *arXiv preprint arXiv:1603.01025*, 2016.

[111] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, "Incremental network quantization: Towards lossless cnns with low-precision weights," *CoRR*, vol. abs/1702.03044, 2017.

[112] C. Leng, Z. Dou, H. Li, S. Zhu, and R. Jin, "Extremely low bit neural network: Squeeze the last bit out with admm," in *Thirty-second aaai conference on artificial intelligence*, 2018.

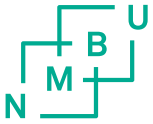
[113] R. J. Cintra, S. Duffner, C. Garcia, and A. Leite, "Low-complexity approximate convolutional neural networks," *IEEE transactions on neural networks and learning systems*, no. 99, pp. 1–12, 2018.

[114] B. McMahan and D. Rao, "Listening to the world improves speech command recognition," in *Thirty-second aaai conference on artificial intelligence*, 2018.

[115] D. S. Park *et al.*, "SpecAugment: A simple data augmentation method for automatic speech recognition," *arXiv preprint arXiv:1904.08779*, 2019.

[116] Y. Wang, A. E. M. Mendez, M. Cartwright, and J. P. Bello, "Active learning for efficient audio annotation and classification with a large amount of unlabeled data," in *ICASSP 2019-2019 ieee international conference on acoustics, speech and signal processing (icassp)*, 2019, pp. 880–884.

[117] E. A. R. Han Wenjing AND Coutinho, "Semi-supervised active learning for sound classification in hybrid learning environments," *PLOS ONE*, vol. 11, no. 9, pp. 1–23, Sep. 2016.



Norges miljø- og biovitenskapelige universitet
Noregs miljø- og biovitskapelege universitet
Norwegian University of Life Sciences

Postboks 5003
NO-1432 Ås
Norway