POLITECNICO DI TORINO

Master degree course in Data Science and Engineering

Master Degree Thesis

Memory-constrained On-device Learning for Smart Infrared Sensors

Supervisors
Pr. Daniele Jahier Pagliari
Pr. Alessio Burrello
Doc. Matteo Risso

Candidate
Tanguy Dugas du Villard
matricola: 321277

Academic Year 2024-2025

Summary

Recent advances in Deep Neural Network (DNN) optimization enable the deployment of these models on resource-constrained embedded devices to perform various tasks. Multiple techniques, such as quantization and pruning, have been successfully applied to reduce the memory, energy, and latency requirements on edge devices. While these improvements enable edge inference, training directly at the edge remains a challenging task, despite its potential benefits for robust and adaptive AI systems. At the same time, low-resolution infrared (IR) cameras have numerous applications as these unobtrusive, privacy-preserving and inexpensive sensors can capture enough information to perform various tasks. By coupling a microprocessor to such IR sensors, it is possible to enable a rich collection of applications that would benefit from DNN inference and training directly on device.

This thesis explores the feasibility of deploying and training a convolutional neural network (CNN) on a severely memory-constrained microcontroller (MCU) coupled to an IR array to classify the pose of a human being filmed by the sensor. The classification algorithm is executed on a bare-metal MCU with less than 32 kB of available memory, where just 14.5 kB can be used to store the model weights and all necessary data. Under such constraints, the usual backpropagation and gradient descent algorithms cannot be applied to update the weights and biases of the model.

In this work, an alternative solution, called Nearest Class Mean (NCM), is explored. Traditional classification algorithms rely on the features extracted in the last layer to compute probabilities for a fixed number of classes. NCMs use a DNN to encode the infrared image in a low-dimensional embedding space. Then, a sample is classified by comparing the distances with the average encoding (so-called prototypes) of each class.

The Convolutional Neural Network (CNN) undergoes a traditional training phase in which the Triplet loss is used to separate the embedding of different

classes. After quantization and a few epochs of quantization-aware training, the model is exported and compiled to be executed at the edge.

The algorithm can reach an accuracy on par with traditional CNN-based classification approaches when classifying classes it has been trained on, with varying accuracy loss when using prototypes of new classes. With a memory footprint fitting the hardware limit, its execution on the edge device is possible. On top of that, the DNN can be trained at the edge, either by updating the prototypes of selected classes to real-world data distributions or by creating a new prototype for a new class. Both without catastrophic forgetting and minimal memory overhead.

This work may lead to the exploration and application of similar algorithms embedded on devices as memory-constrained as 32kB to perform inexpensive, privacy-preserving classifications.

Acknowledgements

I would like to sincerely thank Professor Daniele Pagliari, Professor Alessio Burello, and Matteo Risso for the opportunity to work on this project, their help, support, and supervision throughout the conduct of this work. I would like to thank Francesco Daghero for his time and explanations.

I am grateful to all my teachers at Politecnico di Torino, Centrale Lyon, and La Martinière, without whom I would not be here today.

Many thanks to my parents, who never stopped supporting me even if my work goes over their heads. Thanks to all my friends and family who accompanied me throughout my journey.

Table of Contents

List of Tables

List of Figures

Acronyms

AI Artificial Intelligence.

CNN Convolutional Neural Network.

CPU Central Processing Unit.

DL Deep Learning.

DNN Deep Neural Network.

FPU Floating Point Unit.

FSL Few-Shot Learning.

GPU Graphic Processing Unit.

IR Infrared.

LR Learning Rate.

MAC Multiply–Accumulate.

MCU Micro Controller Unit.

ML Machine Learning.

NAS Neural Architecture Search.

NCM Nearest Class Mean.

NPU Neural Processing Unit.

OP Online Perceptron.

OS Operating System.

PL Prototypical Loss.

QAT Quantization-Aware Training.

SGNB Streaming Gaussian Naive Bayes.

SLDA Streaming Linear Discriminant Analysis.

TL Triplet Loss.

TPU Tensor Processing Unit.

Chapter 1

Introduction

Thanks to the continually improving performance of modern hardware, Deep Learning (DL) algorithms have been successfully applied to perform various tasks, from computer vision and speech recognition over the last decade to the current state-of-the-art generative Artificial Intelligence (AI) algorithms, such as Large Language Models. Because of their important memory, computation and energy requirements, DNNs have been mainly deployed in high-performance servers. Recently, multiple optimization techniques, such as pruning and quantization [1], have been developed to enable the deployment of DNNs on resource-constrained edge devices. These techniques allow for the performance of DNN workloads on data streamed from one or multiple sensors, thereby preserving privacy and reducing bandwidth requirements within the local network and energy consumption. While these optimization methods can reduce the memory, latency and energy footprint of the algorithm, the training algorithm remains offline, on a powerful server, as the traditional backpropagation algorithm cannot be optimized enough to run on severely constrained devices. Multiple alternatives have been tested by researchers and engineers, enabling task-adaptation of DNNs deployed on embedded devices, without the need to rely on servers [2].

In the meantime, Infrared (IR) sensors show multiple applications [3]. Non-obstrusive and energy-efficient, low-resolution arrays are inexpensive and privacy-preserving devices able to capture IR images that can be used to perform various tasks. By coupling a Micro Controller Unit (MCU) to such sensors, face, pose or hand sign recognition tasks can be tackled by an optimized, embedded DNN with minimal economic and energetic cost. Such applications will benefit from on-device learning, as each sensor, environment

and target task may present unique attributes that can even vary over time.

This thesis explains the training, deployment and on-device adaptation of a Convolutional Neural Network (CNN)-based algorithm embedded on a MCU with less than 32 kB of memory. The algorithm receives data from a 16x16 IR thermal array and performs pose recognition by classifying the input into one of five classes. Among the five classes, a selection has been used to train the DNN, while others are completely new to the algorithm and learned online.

Under such constraints, the traditional training algorithm relying on backpropagation and gradient descent cannot be applied because of the required memory overhead. In this document, an alternative solution, called NCM [4], is applied. NCMs are protonets [5]; they rely on the average encoding (produced by a quantized CNN) of each class (prototypes) to classify new samples, as described in Sec. 4.3.1. The prototypes can be updated online with minimal memory overhead and no catastrophic forgetting, usually caused by the training of new classes without reviewing already known classes in an online training setup. This thesis explores the feasibility of executing an NCM algorithm on a constrained device to perform efficient classifications, while being trainable online. The main focuses of this work are:

- Training the CNN to get competitive performance with traditional classification algorithms
- Implementing the algorithm to run on a severely constrained MCU, satisfying memory availability both for data and code instructions.
- Performing training of a new class on the MCU with few streamed data.

After providing the necessary background to understand IR sensors and DNN algorithms, challenges, optimizations and applications in smart sensors (Chapter 2), this thesis reviews some works related to efficient DNNs, smart IR sensors and on-device learning algorithms (Chapter 3). Then, particular attention is given to the methodology, explaining the algorithm, its training and testing (Chapter 4). Finally, the algorithm's performance and memory requirements will be displayed and discussed Chapter 5 before concluding on the work carried out and the potential future works this thesis may lead to (Chapter 6).

Chapter 2

Background

2.1 Infrared array sensors

2.1.1 Technology

IR cameras consist of multiple sensors placed in a grid or a row to produce multiple pixels of IR images. Two technologies dominate the market: thermal detectors and photon detectors.

On one hand, thermal detectors measure the heat transmitted by the IR upon contact. The heat is measured through a bolometer or a thermopile [6], producing a voltage that is amplified, conditioned, and converted into digital signals using an analog-to-digital converter. On the other hand, photon detectors are photodiodes that utilize the photoelectric effect to absorb photons within a target frequency range, producing a current. This current is also conditioned, amplified and converted into voltage and digital signals [6]. Thermal detectors are more sensitive to local temperature and have a slower response time than photon detectors, but are less expensive.

Both sensors will ultimately produce an array of digital signals that can be interpreted as infrared images, the highest values corresponding to high IR lighting, while the lowest correspond to the absence of IR beam. IR beams are produced by a specific device, with customized frequencies, or by any warm body. In the second case, IR sensors can be used to see heat.

In this thesis, a thermal array with a resolution of 16 by 16 is used.

2.1.2 Applications

Unlike normal cameras, thermal IR arrays are low-power, inexpensive and reliable sensors not limited by light conditions. Low-resolution IR cameras are also privacy-preserving (see Fig. 4.1 for example). IR sensors find multiple applications. In the industry, they are used as contactless monitoring tools [7] for predictive maintenance and quality control. In smart homes, to measure occupancy and automatically adjust climate and light [8]. Infrared arrays are also used for face recognition [9] and inside medical devices, for their contactless capacity to localize irregularities and measure heat [10], [11]. Environment monitoring leverages IR arrays to detect wildfires [12] and thermal pollution [13]. Finally, modern cars use IR sensors to detect obstacles and pedestrians [14], helping drivers when the visibility is not optimal.

2.2 Deep Neural Networks

The basic unit of neural networks, the perceptron, was introduced as early as 1958 by Frank Rosenblatt [15]. Mathematicians and computer scientists worked on developing algorithms based on this perceptron until their usage surged with the increase of hardware performance and appearance of challenges, such as the ImageNet recognition [16], in the early 2010s. DNN are the heart of DL, a branch of Machine Learning (ML), and are known to be able to tackle an important variety of tasks. Unlike traditional algorithms, which rely on rules explicitly programmed by humans, ML algorithms, and especially DNNs, leverage the usage of data to learns patterns and relationships between inputs and outputs.

DNNs are today's state-of-the-art algorithms in most AI fields, including generative AI, and are still widely used in computer vision, speech recognition, timeseries analysis and many more. Their ability to handle noisy and complex data without requiring knowledge of the data's nature, combined with their ability to automatically learn, has rocketed their usage across all fields of engineering.

2.2.1 Layers

DNNs are algorithms consisting of multiple steps, called layers, connected to form a network. Multiple types of layers exist; in this thesis, the following

are used:

Input and output layers

Input and output layers are the first and last layers of a DNN. They are tensors of real numbers. The input layer hosts data samples, which are to be turned into outputs to be interpreted. For classification tasks, these outputs are interpreted as probabilities and the index of the highest component of the tensor represents the predicted class. For regression, the tensor does not need to be interpreted, as it represents exactly the output of the algorithm.

Fully connected layers

Fully connected layers, also called Linear layers, are very simple layers composed of nodes inspired by real neurons [17]. Each artificial neuron is connected to every node of the previous layer x_i (for example, every component of the input tensor X) and produces an output $y = \sum_{i=1}^{N} x_i w_i + b$, where b is the bias and w_i are the weights of the neurons, N is the size of the input. Placed side by side to form a layer, the combination of each of the J neurons will produce an output tensor

$$Y = (y_j)_{j=1\dots J} = (\sum_{i=1}^{N} x_i w_{i,j} + b_j)_{j=1\dots J}$$

which can be written as the matrix-tensor product

$$Y = WX + B$$

where W is the matrix of weights and B the tensor of biases.

Convolution

Convolution layers are primarily used to detect patterns within a three-dimensional tensor, such as images with RGB channels, abstracting the position of the pattern in the image. Even if they can be applied to tensors of any dimension, we focus on tridimensional layers. Convolution layers use kernels, which are three-dimensional tensors (made of weights) that are multiplied element-wise with a section of the input. The products are then summed to produce one single value. Then, another part of the input is used

to produce a new value, as shown on Fig. 2.1. The mathematical equation behind convolutions is:

$$O_{x,y,n} = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} \sum_{h=0}^{Z} I(i+x, j+x, h) K_n(i, j, h)$$

where the input tensor I has a dimension of (X, Y, Z) and N convolution kernels K of size (k, k, Z) are used to produce an input tensor O of dimension $(X - k, Y - k, N)$

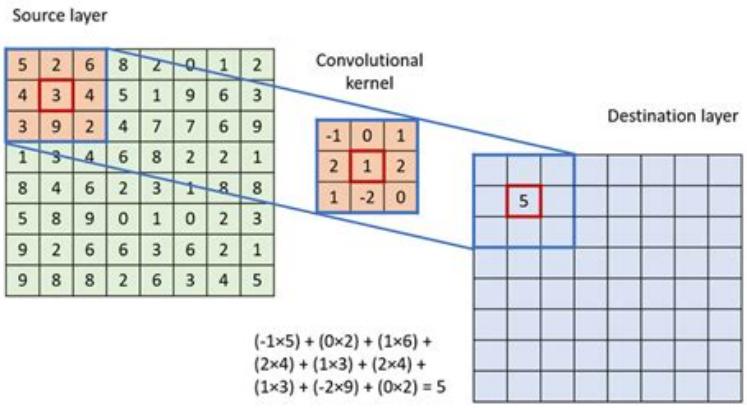


Figure 2.1: Convolution layer (image from [18])

Maxpool

Maxpool layers are used to reduce the size of tensors. Usually applied to three-dimensional tensors, they compute the maximum value of a section of the input, channel-wise.

Activations

Activation layers apply a non-linear function to a tensor. Most of the time, this function is applied component-wise, such as with ReLU, sigmoid, or tanh [19], but some functions are applied to the entire tensor, like softmax, which is used to compute probabilities. These layers are often fused with the preceding layer. Activations are used to learn non-linear relations between inputs and outputs, as all other layers (except max pools) are linear.

Dropout

Dropout layers are layers used to reduce overfitting during training. They randomly set a fraction of the input units to zero at each update step, which prevents the model from relying too heavily on specific neurons. This encourages redundancy and improves generalization performance.

Batch-normalization

Batch-normalization layers are used to normalize the values based on the mean and standard deviation of the input batch (multiple inputs given at the same time to the DNN) [20]. They are used to stabilize and accelerate the training by ensuring the input of the next layer is normalized. During training, given an input x in a batch B, the normalization is performed as

$$\hat{x} = \frac{x - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}, \quad y = \gamma\hat{x} + \beta,$$

where μ_B and σ_B^2 are the mean and variance computed over the batch B, ϵ is a small constant for numerical stability, and γ, β are learnable parameters.

To allow consistent predictions at inference, running estimates of the mean and variance are maintained during training, typically with exponential moving averages:

$$\mu \leftarrow (1 - \alpha)\,\mu + \alpha\,\mu_B, \quad \sigma^2 \leftarrow (1 - \alpha)\,\sigma^2 + \alpha\,\sigma_B^2,$$

where α is a momentum parameter (usually close to 0.1).

During inference, these running estimates are used instead of batch statistics:

$$\hat{x} = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}, \quad y = \gamma\hat{x} + \beta.$$

These estimates are used to ensure consistency throughout multiple inputs. In reality, during inference, the batch-normalization layer can be fused with the preceding fully connected or convolutional layer by adjusting its bias and weights.

2.2.2 Training

For DNNs to perform, they need to be trained with known, controlled data. During this process, the model will learn how to map inputs and outputs

by updating the weights and biases of its layers to make the predictions as reliable as possible. The well-known algorithm used to update the DNN is called backpropagation and relies on the gradient descent algorithm and the chain rule [21]. To produce a gradient, a loss function L is used to give an image of the error the DNN is producing. Examples of losses are the cross-entropy loss [22] for classification tasks and the mean absolute error and mean square error for regression tasks. Because each function used in each layer is known and differentiable, it is possible to compute, for every parameter (weights and biases) θ, the gradient of L with respect to θ, $\frac{\partial L}{\partial \theta}$. Using this gradient, each weight is then updated with the following equation:

$$\theta \leftarrow \theta - \eta \frac{\partial L}{\partial \theta}$$

where η is the learning rate, a hyperparameter with an order of magnitude usually below $1e-3$. Different update equations can be used, called optimizers, and implement momentum and/or regularization, such as ADAM or ADAGRAD [23]. This algorithm is repeated until the loss reaches a minimum (early-stopping, in this case, the loss is measured on a dataset unused during training) or until a maximum number of iterations (epochs) is reached. The performance of the DNN is usually measured in a more human-readable way, like the accuracy for classification tasks or the mean square error for regression. Other metrics can be used depending on the specific case.

2.2.3 Applications

Deep learning founds a large number of applications. First, computer vision relies heavily on CNN [24] to classify images, identify objects, identify images or locate objects on images. Those algorithms are massively applied in healthcare [25] to diagnose diseases in early stage, in autonomous driving [26] to detect cars, pedestrians and signs, in manufacturing [27] etc. DNNs are also widely used in speech recognition and natural language processing [28], primarily used by voice assistants and are today fundamental in generative AI and its applications. Other applications include anomaly detection [29], fraud detection [30], scientific research in many fields as biology [31] and history [32], recommendation systems [33], predictive maintenance [34] and trading [35].

2.2.4 Challenges

Despite being very powerful, DNNs present many challenges described below.

Overfitting

Overfitting is the phenomenon in which the model not only learn the pattern but also the noise and outliers, resulting in strong performances on training data but poor ones on data never seen by the model during training. Overfitting occurs when the model is too complex for the assigned task and can be addressed through early stopping, regularization, and dropout [36].

Data Quality

Data quality is very important in deep learning, as it needs to represent as accurately as possible the inputs used during the usage of the DNN. If the training data are noisy, wrongly labeled, biased, restricted to a subset of real tasks, or imbalanced, the model's accuracy will not be as good as it could be.

Hyperparameters and layer choice

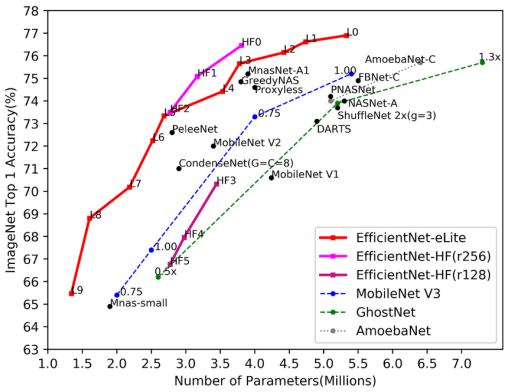
Hyperparameters and layers are to be taken seriously as they can change the behavior or training of the model. For example, a too low learning rate will lead to slow learning and a performance being trapped close to a local minimum, while a learning rate too high can make the model struggle to reach the minimum. The number of neurons in each layer, the size of kernels, the choice of activation and the number of layers themselves can be seen as hyperparameters that should be tuned.

Memory

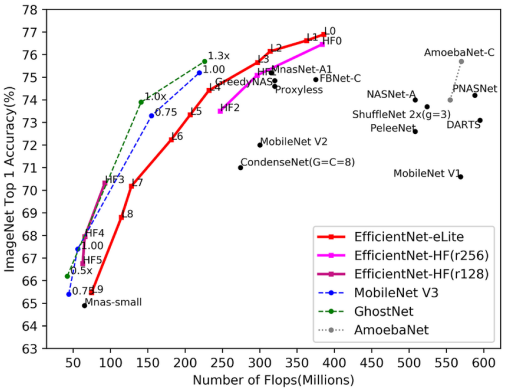
The Memory footprint of a deep learning model tends to be very high. For example, a fully connected layer with an input of size I and an output of size O contains a matrix of size $I \times O$ (for the weights) and a vector of size O (for the biases), which can reach thousands of parameters. With $I = 0 = 64$, the layer contains 4160 parameters, which is approximately 16 kB. Recent best models used to perform object classification on the Imagenet dataset [16] reach millions of parameters, a few of them are shown on Fig. 2.2a. On top of these parameters, memory is necessary to store intermediate activations and store and compute gradients.

Energy consumption and latency

Energy consumption, similar to memory, can be very high for big models. The same fully connected layer as previously will execute approximately $(2 \times I + 1) \times O$ floating-point operations (FLOP) to compute the output (which is 8256 operations). More complex models can reach hundreds of millions of FLOP, as shown on Fig. 2.2b. While one single flop consumes energy, it also needs time. Finally, the data used to compute the output needs to be moved from the memory to the cache and registers, which also takes time. Running a DNN on a low-power MCU, generally fuelled by a battery, or with a high frequency, is a challenge. Data transfer and frequency can be solved by computing outputs in batches, layer by layer, or by using specific accelerated hardware.



(a) Number of parameters for various DNN



(b) Number of FLOPS for various DNN

Figure 2.2: Memory and computation requirement for various DNN (images from [37])

2.2.5 Optimizations

To tackle the memory and resource challenges, several tricks can be implemented.

Data reuse and batching

While computation can be the limiting factor for latency, data transfer is also very costly, both in terms of energy and time. In fact, transferring data from

the memory to the registers to compute an operation is hundreds of times less efficient than using a datum already stored in a register [38]. A solution involves using temporal layer tiling [39], an optimized method for moving data between different storage locations, from the processor's registers to caches and main memory. Some hardware optimization implements this [40]

Another solution that is widely used during training but can be problematic during inference is the usage of batching. In this case, instead of performing a forward pass on inputs one by one, all inputs are used together and each layer is used only once. This is very helpful as it allows the layers to be loaded only once for the whole batch. During inference, this method assumes that multiple inputs are collected, which might not be true for systems requiring real-time computation, like autonomous driving or biosignal processing. In those cases, if we wait 1 second to collect samples, the result for the first element of the batch will be 1 second late, which can be catastrophic as decisions must be taken quickly.

Hardware optimization

Most of the computations realised by DNN are Multiply–Accumulate (MAC) operations, which are the operations used to compute a matrix-vector or matrix-matrix product. Most modern Central Processing Units (CPUs) contains a few MAC units, specialized in MAC operations; however, Graphic Processing Units (GPUs) and Tensor Processing Units (TPUs) can have thousands of them [41]. They are optimized to perform vector-vector product (GPU) and matrix-matrix product (TPU) as fast as possible and leverage parallelization to compute as many operations as possible. These processors also have optimized memory configurations and a very high memory bandwidth to limit the data transfer cost [42]. Current works also focus on accelerators for edge devices [43]. Other hardware accelerations can be used together with some of the following software optimisations, as compatibility between both may be required.

Convolutions as GEMM and depthwise-pointwise

Convolution layers are computationally expensive but also require the loading and unloading of a lot of data if not implemented correctly. One way to reduce this phenomenon is by organising and duplicating the data in memory in such a way that the convolution becomes a matrix-matrix product, which,

as seen previously, can be optimized in multiple ways [44] [45]. This process is called im2col.

The second optimisation can be done by replacing a normal convolution with spatially separable convolutions [46]. Spatial separable convolutions can be used if the convolution's kernel can be written as a product of two 1D tensors, for example,

$$\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix} \times \begin{pmatrix} -1 & 0 & 1 \end{pmatrix}$$

By multiplying the input by the unidimensional tensors instead of the matrix, you can save some computation while getting the same result. Moreover, less memory is necessary to store the kernel. Unfortunately, not all matrices can be written as a product of two vectors. In this case, it is possible to separate the convolution into a depthwise and a pointwise convolution. The depthwise convolution will create an output having the same number of channels as the inputs, then the pointwise convolution will use this output to create an image with the expected number of channels. Doing this, the input is only really transformed once [47].

Double buffer

A very simple, effective way to reduce the memory footprint of a DNN is to allocate only the memory necessary to store the two largest activations. One part of this is used to store the output of odd layers, and is then used to be the input of the even layers. The other part does the opposite. While this method is efficient, it has an important drawback: the activations are overwritten and thus cannot be used to compute a gradient, making it impossible to perform the backpropagation algorithm to train the model.

Neural Architecture Search

Neural Architecture Search (NAS) is a technique used to optimize the choice of hyperparameters and layers [48]. As explained in Sec. 2.2.4, choosing the best hyperparameters and architecture can be difficult. Usually, this choice is made by exploration, trials and errors, by comparing the performance (through the loss or another metric) of different combinations of parameters. While this technique works to find the best-performing model, it does not

consider the complexity, energy consumption, latency or memory footprint of the model. As [48] shows, one way to consider it is by adding a cost term to the metric. For example, instead of relying on accuracy for the final choice of model, we can rely on the difference between the accuracy and the number of parameters, or the average execution time plus the F1-score, with some scaling factors to balance the importance of both components.

NAS can also allow the engineer to find the best model more efficiently. In reality, finding the best architecture is not easy and every layer can be replaced with another, some layers can be added, some can be removed, the number of neurons can be changed etc. A solution is to use multiple options at the same time, and to select the best one based on a parameter that will be trained. This parameter, called a gate, will tend to 0 if its corresponding layer is not better than another layer whose gate will tend to 1. This technique can be used with every other technique presented, including the cost term in loss functions.

Pruning

Pruning is a technique used to reduce the number of weights and connections in a layer. Pruning techniques are multiple, as DNNs are purposely too complex for the task it is assigned, multiple parts of the algorithm may be skipped with minimal loss of performance [49].

The three main pruning algorithms consist of replacing selected weights by 0, skipping computations by disconnecting neurons or skipping the computation of activations that are usually equal to 0. This occurs regularly when using a ReLU activation.

Pruning weights and connections has multiple benefits. First, using sparse matrices can be beneficial in terms of memory. Then, skipping computation makes the model faster. However, we need the hardware to be able to manage matrix-vector products when the matrix has a sparse representation.

There are multiple ways to prune weights. Replacing values inside of a tensor can be done in a structured way, by removing blocks of values, an unstructured way, by removing values without taking into account their position in the matrix, or by using a balanced structure to keep the number of non-removed values constant across blocks, rows, or columns.

The weights to be pruned can be determined with multiple criteria. The simplest is magnitude, as we can expect that replacing values close to zero with exact zeros will have minimal impact on the performance. For regression

tasks, a very small difference in the output may appear, but for classification tasks, it is very unlikely. If the probabilities are slightly different, the class of maximum probability is unlikely to change. The weights can also be pruned based on their impact on performance [49] or on latency [50].

The pruning process can be scheduled in multiple ways. One option is to prune a given percentage of weights after the final training epoch. This "train-then-prune" algorithm can lead to performance losses and a few training epochs can be added, in which the remaining weights are updated. The pruning can also be done progressively, according to a pruning schedule, at the end of each epoch [49].

Quantization

The most important optimization technique discussed in this document is quantization, which aims at using fewer bits to represent each weight and bias.

Weights are usually represented by 32 bits (4 Bytes, float32 format) but can be approximated with 16 bits (float16 format). Applying this approximation to all layers will divide the memory footprint and data transfer latency and energy consumption of the whole DNN by approximately 2. It can also be applied to activations and does not lead to a significant performance loss. This float format can be used during training and inference [51], but representing weights in an 8-bit (or even less [52], [53]) format is also possible.

Focusing on integer format, which is the main format usually supported by edge devices, representing weights and activations this way allows an important memory footprint reduction, as well as reduced latency and energy consumption. In this format, we represent weights by integers in the range $[0, 255]$ or $[-128, 127]$. In reality, weights can have a much different range and quantizing does not correspond to rounding. A few different transformations can be used to efficiently transform a 32-bit floating-point value into an 8-bit integer

The simplest quantization process is called affine quantization [54], where an affine transformation is applied to the real weights to fit the expected range, before being clamped to the expected range, as follows:

$$\tilde{A} = \text{clamp}(\lfloor A/s \rfloor + z, m, M)$$

where A is a value to be quantized and $\tilde{A}$ its quantized version, m and M the minimum and maximum of the range, s a scale factor and z an integer called

zero-point, used to align the range of A/s to $[m, M]$. Then, the dequantized $\bar{A}$ equivalent to $\tilde{A}$ can be computed as

$$\bar{A} = s \times (\tilde{A} - z)$$

which in theory should be as close as possible to A.

In practice, the statistical distribution of the weights is symmetric, which means that the signed range $[-128, 127]$ is used to store the quantized weights, setting z to 0 and s to $\max(|W|)/127$, where W is a set of weights. Using a single scale factor for the whole layer allows for limiting the number of parameters, but the actual range of the weight can be completely different between two channels of the same convolutional layers. For this reason, multiple scale factors can be used for different blocks, rows or columns of a fully connected layer's weight matrix or for different channels of a convolution matrix. In this case, the clamping is actually unnecessary, as the scale factor is made to map the largest weight to 127.

Activations, however, are using asymmetric quantization. Focusing on ReLU functions, the output of a layer is always positive, which means we can set z to 0 and s to $\max(X)/255$, where $\max(X)$ is the maximum value an activation can assume. Unlike for weights, this $\max(X)$ is not known as activations are not stored inside the DNN, only weights are. To compute this maximum, a batch of representative examples is needed. Outliers can still appear and the clamping will restrict the value.

In fact, outliers can also be part of the set of representative inputs. For weights and activations, the scale can be set to a lower value, which will force the highest of them to be clamped but can actually improve performance, as a smaller scale implies a greater precision for weights and activations closer to 0. For this reason, the scale can be computed, or learned, to minimize the approximation error $\sum_{w \in W} |w - \bar{w}|$, where W is the set of all weights of a layer. The same principle applies for activations [54].

The quantization described previously will be more efficient for uniformly distributed weights, but in reality, weights usually follow a Gaussian distribution, which means that most weights close to 0 will get approximated by the same integer, while most of the highest half of $[0, 256]$ will not be assigned to any weight, which is a suboptimal approach. Mapping the weight distribution to $[-128, 127]$ can be done in a more optimized way to spread the weight in the range and achieve better precision [55] [56] using the logarithm, which can be computed on edge devices with a scale and a shift.

As for pruning, quantizing a DNN can be done after or during the training. On one hand, quantizing after training consists only of computing the new weights, scale and zero point (or other parameters necessary for the algorithm), based on the current value of the weights and on a representative sample of inputs. On the other hand, quantizing during training consists of a training phase, a quantization and another training phase, called Quantization-Aware Training (QAT). In this third phase, the backpropagation algorithm cannot be applied as usual. The gradient descent used to update the weights assumes the weights can take any value, not only integers. This is solved by using fake quantization. During the forward pass, an approximation of the real value:

$$\bar{W} = s \times (\tilde{W} - z)$$

During the forward pass, the gradient of $\bar{W}$ is propagated to W [54] (removing the rounding) to obtain a new value for W, then the scales and zero-point are recomputed, or can also be trained.

Weight clustering

Weight clustering is a technique that can be used alongside weight pruning and quantization to reduce the size of an DNN, but its application may decrease the DNN's performance. The clustering algorithm consists of grouping weights based on the proximity of their values. Then, the tensor of weights is turned into a tensor of groups, where each weight is replaced by the index of the group it belongs to, and a lookup table is used to map the index to the real value. Each weight is approximated by a value common to the whole group.

Two techniques can be used. The first consists of spreading uniformly the K centroids in the weight range, then regrouping all the weights closer to one centroid together. The other consists of using the K-means algorithm [57]. In practice, the first is used to initialize the centroids before applying the K-means algorithm. To represent one index, only $\lceil log_2(K) \rceil$ bits are necessary, when 32 were necessary to represent each weight in the float32 format, allowing a high compression ratio.

As for quantization and pruning, weight clustering can be done after training, or during training. In both cases, during the forward pass, the weights are replaced by the centroid of their cluster, using the lookup table, which is less efficient computationally and in terms of performance, but allows

a high compression. Improving the performance can be achieved through clustering-aware training. In this case, the forward pass is performed as described above, while the backward pass differs from the standard algorithm only in the update of the weights. Instead of relying on the gradient of each weight, the gradients of all weights are summed and the sum is used to update the centroid.

The centroids may be quantized after training, allowing further compression rates.

2.3 Smart Sensor Hardware Platforms

Smart sensors are hardware devices consisting of a computation unit connected to one or multiple sensors and a communication interface. The smart sensor records data from its environment. The data are transformed and transmitted to a control unit that communicates with multiple sensors to make decisions.

Unlike normal sensors, which collect data and transmit them to a cloud server, some computations are performed by the embedded algorithm, which can be a DNN, running on an edge device. Moving the DNN to the edge device offers multiple benefits, including improved privacy, reduced network load, lower latency, and lower energy consumption. As an example, a face recognition system requires a camera filming the face of a person. Transmitting the full face image to a server can lead to multiple privacy issues and requires the transmission of up to megabytes of data. While DNN inferences consume energy, data transmission is known to consume even more. Finally, relying on a server to have a response on the face recognition task can take time. Transmitting the data and waiting for an answer can lead to a very high latency.

DNN inference is not always possible at the edge, as complex DNNs can require billions of operations. For low-frequency processors, it can take seconds to compute. Processors might also not have the required hardware to compute divisions or MAC operations for efficient inferences, or the available memory can be too limited.

Edge devices are various; some are only slightly constrained, like smartphones and single-board computers, with GigaBytes of available memory, high computation capabilities and a custom Operating System (OS) for performance. On the opposite edge of the spectrum, some MCU only embed

a few computation units and kiloBytes of memory, without any OS. In between, some companies are developing specialized inference devices like Neural Processing Units (NPUs) or embedded TPUs.

2.4 On-device Continual Learning

In most applications involving DNN, the DNN is trained offline. A powerful machine with one or multiple GPU is used to train a model on a large amount of data and without memory and energy constraints. The DNN is then either deployed to an unrestrained cloud platform or optimized and deployed on an edge device with limited memory and energy consumption requirements. In both cases, the DNN can be used as-is, but some applications require the algorithm to be fine-tuned after deployment. For example, a speech recognition algorithm needs to adapt to the user's accent, voice, or vocabulary [58], which differs from the one used in training data.

2.4.1 Continual Learning

In an online learning setup, the training data are streamed to the DNN for training one by one, or by very small groups, and are only seen once. It is significantly different than offline learning, where the data are stored in the memory of the computer to be used multiple times, and can be provided in batches. Continual learning is the application of online learning after deployment of the DNN. The new training data comes from the model's real-world environment, arriving as a stream of user interactions, sensor readings, or contextual information. This stream can have a distribution different than the distribution of the training data, and can even drift over time.

The data can also come in an unbalanced stream, as the real world might change more slowly than the data collection frequency. Unbalanced datasets can lead to catastrophic forgetting [59], as the DNN will focus on being correct on the last data. Some techniques, such as replay [60], which consists of storing data samples and updating the DNN only with balanced data, can be applied.

Finally, the training data can be scarce, either because the data collection process is low-frequency, or because it can be time, resource or energy consuming. This is called FSL. A setup in which the training data may not

be representative of the real distribution.

2.4.2 On-device learning

Executing continual learning algorithms on constrained devices raises several challenges due to computation limitations and low-memory availability. First, the lack of memory can make it impossible to store multiple input data, forcing the batches to be small and blocking the usage of the replay method. The lack of memory can also lead to the need to discard activations, by using the double buffer introduced in Sec. 2.2.5, which makes the computation of gradients impossible. Even if these activations are stored, the computation of the gradient is memory-intensive, and even more so if the training is performed with batching. Finally, both the forward and backward passes are computationally intensive. Some MCU have a clock frequency thousands, if not millions of times lower than the current CPUs and GPUs, in some cases, the training algorithm could require more time than the delay between two inputs. The training algorithm will also consume energy that can be scarce if the edge device is battery-supplied.

Chapter 3

Related Works

3.1 Deploying Efficient Deep Neural Networks

For more than a decade, the growth in the performance of DNNs has led to the development of an increasing number of models meant to be executed on edge, constrained devices. To fulfill the memory and computation constraints, multiple methods have been employed, the earliest one including quantization, pruning, and other DNN optimizations [1], which achieve extreme compression of well-known architectures, thereby reducing the computation and memory requirements of running such algorithms, with little to no accuracy loss. Other techniques include distillation, in which a powerful model trains a lightweight model to perform a similar task [61], or deep neural architecture search [48], where the architecture itself is encoded with trainable parameters, and in which the training loss includes criteria such as memory or latency. Some frameworks have been developed to help data scientists run DNN on edge devices, such as LiteRT (formerly known as TensorFlow Lite) [62], ExecuTorch [63] and OpenVINO [64]. Other research focuses on a family of DNNs, called EfficientNet [37], to reduce the complexity of DNNs without impacting the model's performance.

These techniques have been successfully applied to run CNNs for computer vision applications on smartphones, such as the multiple versions of MobileNets [65] [66] [67] [68], or on single-board PC like Raspberry Pis [69].

CNNs have been recently deployed to severly constrained MCUs for human activity recognition [70] and ventricular fibrillation detection [71]. Some other algorithms are deployed on constrained hardware modified to optimize DNN inference, for eye detection [72] and smart cameras [73], for example.

Deploying efficient DNNs is a preferred choice in IoT applications, as it has been successfully applied in smart homes [74] for energy management and environment monitoring [75].

3.2 IR Sensors Applications With Deep Neural Networks

IR sensors have been used for decades to record thermal images, as a communication support and for health monitoring. Recently, IR images have been coupled with DNN to perform various health monitoring tasks [76] [77] and tumor detection [78]. CNNs have also been applied to analyze soil [79] and materials [80] as well as face recognition [81] or defect detection [82], by performing classification tasks on infrared images or infrared spectra.

3.3 On-device learning

Training DNN on constrained edge devices poses a challenge even more complex than using an embedded DNN to produce inferences, as updating the weights requires significant memory and computational capabilities, and the use of non-batched training input can lead to catastrophic forgetting [59] (see Sec. 3.3.1). Over the last few years, multiple algorithms have been successfully tested for training embedded DNNs [2].

3.3.1 Updating efficiently the model's weights

Some of the main techniques differ slightly from the usual backpropagation and gradient descent. Recently, [83], [84], [85] implemented sparse update algorithms to update a few selected layers, or the biases only. With this method, memory requirements and gradient computations are decreased, which enables training on constrained devices. These methods need to be coupled with replay [60] to avoid catastrophic forgetting in a FSL setup, or other techniques, as [86] proposes restoring stochastically a subset of the weights after training.

Other alternative algorithms have been proposed to avoid the memory or computation overhead of backpropagation. Sparse approximation [87] limits the memory overhead but requires extensive computation. Multiplexed

Gradient Descent [88] fastens the training of DNNs with a first-order approximation of gradients, while [89] uses a zeroth-order approximation and [90] uses both. Rep-Net [91] uses transfer learning to carefully update the embedded DNN. Minilearn [92] applies on-device learning, achieves important memory and latency requirement reductions, and even improves the model performance.

Other techniques can be applied to avoid catastrophic forgetting. [93] leverages meta-learning while [94] explores self-synthesized rehearsal for Large Language Models and [95], similarly to replay [60], continuously retrains weights with already seen (compressed) inputs.

3.3.2 Protonets

Alternative algorithms rely on prototypes, which are average representations of an input class [5]. In the previous examples, the DNN weights were updated to adapt the model to new classes. Now, the model will not be updated by itself, but instead, will be used to compute prototypes, an average representation of each class in a low-dimensional space. The prototypes will be used to compute the class. All Protonets are efficient algorithms to tackle the challenges of device constraints, as the update of the prototype requires less memory than the update of the model weights, and do not suffer from catastrophic forgetting.

Four different protonets have been benchmarked by [5] in FSL setup: Online Perceptron (OP), NCM, Streaming Linear Discriminant Analysis (SLDA), Streaming Gaussian Naive Bayes (SGNB).

The first two algorithms directly use the prototypes and their distances to the representation of a sample to classify it. Distances can also be normalized to produce probabilities. They differ in the way the prototypes are trained.

In OP, an input x of class i will update the prototype of class i P_i by applying

$$P_i \leftarrow P_i + f(x)$$

while the prototype P_i^* the furthest from $f(x)$ will be updated with

$$P_i^* \leftarrow P_i^* - f(x)$$

The NCM algorithm is even simpler, as the prototype is the average of all the embeddings of inputs belonging to the same classes, it is further explained Sec. 4.3.1. The NCM algorithm has been used for several years [4]. While

being more accurate than OP [5], NCM requires the number of prototypes to be stored and the possibility to compute accurate divisions.

SLDA and SGNB compute prototypes the same way NCM does, but also rely on covariance matrices to classify embeddings. Alongside the prototypes p_i, the SLDA algorithm stores a covariance matrix Σ updated as

$$\Sigma \leftarrow \frac{c_i \Sigma_i + \Delta}{c_i + 1}$$

with

$$\Delta = \frac{c_i (f(x) - p_i)(f(x) - p_i)^T}{c_i + 1}$$

with x an input of class i and c_i the count of inputs already used.

The matrix and prototypes are then used to produce a vector of probabilities, according to [96], which requires inverting the matrix Σ

The SGNB is simpler, as it does not store a covariance matrix, but only the variances (Σ's diagonal).

In both cases, the matrix can be updated at each sample or computed once. In the first case, the inverse matrix needs to be recomputed after each update, which can be computationally expensive, while in the second case, the matrix can be computed offline. In SGNB, the inversion is easier as it only involves inverting each value on the diagonal. Compared to NCM, more memory and computation are required, but performance can be better [5].

Chapter 4

Methods

This chapter focuses on describing the data (Sec. 4.1), device Sec. 4.2, algorithms (Sec. 4.3), the training (Sec. 4.3.7) and testing of the DNN (Sec. 4.3.8), and the implementation of the NCM running on the edge device. The algorithm performance will be compared to and a traditional classifier for accuracy and the replay algorithm [60] for continual learning, explained in Sec. 4.4,

4.1 Dataset

The dataset used in this work consists of images recorded with the sensor itself, to which a label representing the pose of the person being filmed by the sensor is assigned. These poses, which are the $N = 5$ classes we want our algorithm to predict, are the following:

1. Empty (the sensor is not filming anyone)
2. Standing (someone is standing in front of the sensor)
3. Right arm (someone is standing and raising the right arm)
4. Both arms (someone is standing and raising both arms)
5. Left arm (someone is standing and raising the left arm)

A sample from each class is shown on Fig. 4.1.

These data have been collected during multiple sessions of recording, in different (but similar) environments with different ambient temperatures,

leading to a small distribution difference between both two datasets that can be visualized by showing the distribution of the prototypes' and embeddings' components.

In all recordings, the only hot object filmed is the person. The set of training data and the set of validation data come from distinct recording sessions, to mimic a real-life application, where the data used to train the DNN comes from a different source than the data that will be classified or used for training the NCM online.

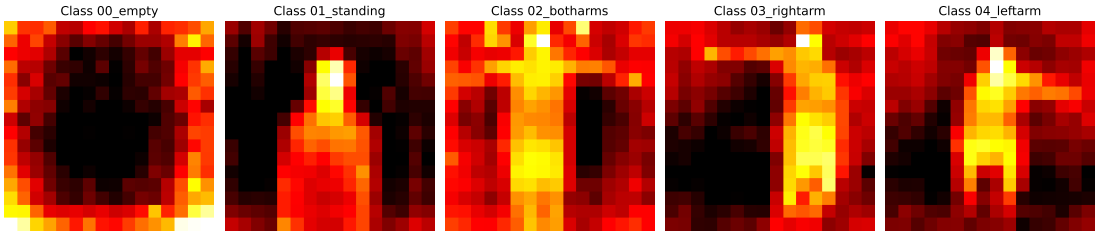


Figure 4.1: A sample from each class

4.2 Target device

The system on which the algorithm is tested, called MAUPITI, consists of a customized IBEX core [97] and an array of 16x16 TMOS thermal sensors measuring IR radiations at a frequency of 10 frames per second [98]. The core has two 16 kB memories, one for instructions and one for data, from which respectively 3.5kB and 1.5kB are used to extract the data from the sensor, leaving 27kB available, 14.5kB for storing weights, biases, activations, etc. A second core plays an intermediary role, receiving the code to be executed on the IBEX core and writing its memory. This core also reads data from the IBEX core's memory to send them to the computer.

4.3 Algorithm

The selected algorithm to classify inputs and be on-device trainable with high memory constraints is the NCM. It is one of the protonets presented in

Sec. 3.3.2

4.3.1 Nearest Class Mean Classifier

The NCM is a Protonet, as described by [5]. Instead of relying directly on the output of a DNN to classify inputs, we use the output as a representation of the input in a space of low dimension and classify the input based on distances to an average representation of each class (called a prototype). Mathematically, the equations are very straightforward. For each class c, the prototype P_c is initialized to the null tensor 0_p, and a counter n_c is set to 0. Then, for each new input x of class c, the prototype is updated as

$$P_c \leftarrow \frac{P_c n_c + f(x)}{n_c + 1}$$

and

$$n_c \leftarrow n_c + 1$$

Which means that P_c is the average tensor of all encoded inputs $f(x)$.

To classify a new input x, we find the closest prototype to the encoding of the input:

$$c^* = \underset{c=1...N}{\operatorname{argmin}}(\operatorname{dist}(P_c, f(x))$$

4.3.2 Distance

The NCM requires a distance metric to classify samples, but also to compute the loss. Two distances have been tried:

1. The Euclidean distance

$$\operatorname{dist}(x, y) = (\sum_{i=1}^{N}(x_i - y_i)^2)^{0.5}$$

   For this distance, the square root can be omitted when looking for the argmin.

2. The Angular distance

$$\operatorname{dist}(x, y) = 1 - \frac{x.y}{\|x\| \, \|y\|}$$

It is in the range [0, 1]. When classifying, the computation can be simplified as

$$c^* = \underset{c=1...N}{\operatorname{argmax}} \frac{P_c.f(x)}{\|P_c\|}$$

which can be computed with integers by multiplying the numerator by a high number, for example 2^{15}. This can be implemented in two ways:

- The first algorithm stores only the norms and the prototypes and applies exactly the equation shown above.
- The second implementation stores $P'_c = \frac{2^{15}P_c}{\|P_c\|}$ in memory and uses it to fasten the computation, as

$$c^* = \underset{c=1...N}{\operatorname{argmax}}(P'_c.f(x))$$

which does not require a division to be performed.

3. The Manhattan distance:

$$\text{dist}(x, y) = \sum_{i=1}^{N} |x_i - y_i|$$

4.3.3 The Deep Neural Network

The DNN used for this experiment has been defined by [99] and is made of the following layers:

0. The input layer hosts a tensor of size 16x16, representing an infrared image. Each value is an integer between 0 and 255.

1. A convolutional layer with 8 filters and a kernel of size 3x3, applied without padding and with a stride of 1 in both directions. Its output is a 14x14x8 tensor. It contains 80 parameters.

2. A batch-normalization layer, followed by a ReLU activation. It contains 16 parameters.

3. A convolutional layer with 8 filters and a kernel of size 3x3, applied without padding and with a stride of 1 in both directions. Its output is a 12x12x8 tensor. It contains 540 parameters.

4. A batch-normalization layer, followed by a ReLU activation. It contains 16 parameters.

5. A maxpool layer with a pool size of 2x2, no padding and a stride of 2. Its output is a 6x6x8 tensor. A Maxpool does not contain any parameters.

6. A convolutional layer with 8 filters and a kernel of size 3x3, applied without padding and with a stride of 1 in both directions. Its output is a 4x4x8 tensor. It contains 540 parameters.

7. A batch-normalization layer, followed by a ReLU activation. It contains 16 parameters.

8. A maxpool layer with a pool size of 2x2, no padding and a stride of 2. Its output is a 2x2x8 tensor. A Maxpool does not contain any parameters. The output is flattened to become a 1D-tensor of size 32.

9. A fully connected layer whose input is a tensor of size 32 and whose output is the representation of the infrared image in the latent space, used to compute the prototypes (if used to train the NCM) or to be compared with them to classify the input (if used for inference). This representation is a tensor of size p, the prototype size, which is a hyperparameter. The layer contains $33p$ parameters.

Overall, the DNN contains $1208 + 33p$ parameters. The largest activations are the outputs of the first two batch-normalization layers, with respectively 1568 and 1152 components. Using the double buffer technique during inference, removing the 48 parameters of the batch-normalization layers fused with the preceding convolutional layers, and considering the $5(p + 1)$ values reserved for the prototypes and counts, the algorithm finally requires $3885 + 38p$ variables.

4.3.4 Optimization

For the algorithm to run on the MCU, it needs to be optimized. To do so, I used PLiNIO [100], an open-source project developed by Politecnico di Torino, and used to perform quantization and Neural Architecture Search. After quantizing each layer with a PACT quantizer, some C files are created, containing the code to execute inferences and training with the DNN. These files are then compiled and are ready to be executed on the smart sensor.

4.3.5 Loss

During the training phases described below, one of the two following losses can be used. The Prototypical Loss (PL) and the Triplet Loss (TL) [101] are both based on distances. The distance used can be either Euclidean or Angular.

Prototypical Loss

The prototypical loss is very straightforward, as it is analogous to the cross-entropy loss for normal classification. The main difference is in the usage of part of the batch to compute prototypes; the other part is used to compute the loss. For each of the $N - k$ classes, the batch contains exactly S support samples, used to compute the prototypes, and Q query samples. The prototype of class c is computed as:

$$P_c = \frac{1}{S} \sum_{s=1}^{S} f(x_{c,s})$$

Then, the loss is computed over the $Q \times M$ query samples as

$$L = -\frac{1}{(N-k)Q} \sum_{q=1}^{Q} \sum_{c=1}^{N-k} \log \frac{\exp(\text{dist}(f(x_{c,q}, P_c))}{\sum_{i=1}^{N-k} \exp(\text{dist}(f(x_{c,q}, P_i))}$$

Triplet Loss

The triplet loss is computed based on triplets. It's an improvement over the prototypical loss that explicitly tries to separate samples from different classes. Here, the batch also contains the exact same number of samples in each class. The triplet is selected as follows:

1. For each class c
2. For each combination a, p of samples from class c (the positive samples)
3. For each sample n from another class (the negative sample)
4. If the distance between $f(a)$ and $f(p)$ is lower than the distance between $f(a)$ and the negative sample $f(n)$ (minus the margin m), mark the negative sample as a candidate

5. Select one random negative sample $f(n^*)$ among all candidates, if None, the pair $f(a), f(p)$ is dismissed

Then, the loss is computed over the N_t triplets with

$$L = \frac{1}{N_t} \sum_{i=1}^{N} \max(0, \text{dist}(f(a), f(p)) - \text{dist}(f(a), f(n)) + m)$$

4.3.6 Quantizer

The three linear quantization algorithms used to quantize different parts of the DNN are presented below.

MinMax

The MinMaxWeight is a very simple linear quantizer used to quantize weights. To each channel of a convolution layer, or each neuron of a fully connected layer, a scale s is assigned. The real weights W are quantized to be stored as

$$\tilde{W} = \lfloor s \times W \rfloor$$

where $\lfloor a \rfloor$ is the integer part of a. Then, during the forward pass of an input, an estimation of the real weight, $\tilde{W}/s$ is used. As it is slightly different than the real value, some performance can be lost. The value of s depends entirely on the real values of W and the precision used to store the weights, which is 8 bits for our application.

PACT

The PACT Quantizer [102] has two or three parameters: a scale s and one or two clamping values m and M, which represent the maximum value an input can assume. For unsigned inputs, the lower clamping value is set to 0. The output is computed as

$$y = \lfloor s \times \text{clamp}(x, m, M) \rfloor$$

where $\text{clamp}(x, m, M)$ is the clamped value of x, which is m if $x < m$, M if $x > M$ and x otherwise. m (if not set to 0) and M are trainable parameters. s can be computed from m, M and the number n of bits used to represent a value, as $2^n = (m - M)/s$. PACT is used to quantize activations, which

are inputs of hidden layers and also the outputs of the following layers. In our case, $n = 8$. They are very close to the MinMaxWeight presented above, but differ only in the usage of clamping. The reason behind this clamping is that we do not know the maximum and minimum values that an activation can assume, while for weights, it is known.

Bias Quantizer

The Bias Quantizer is as simple as the weight quantizer. The biases are stored with a precision of 32 bits and the corresponding scale is equal to the product of the scale of the input activation and the weights.

Quantized fully connected layer equation

For a complete, quantized, fully connected layer, if s_x, s_w and s_y are respectively the scales of input, weights, and output, and m_i, M_i, m_o and M_o are the clamp values of the input and output PACT quantizers, we obtain

$$s_y \text{clamp}(\tilde{Y}, m_o, M_o) = s_x s_w (\tilde{W}.\text{clamp}(\tilde{X}, m_i, M_i) + \tilde{B})$$

where $\tilde{A}$ is the quantized version of any tensor A.

4.3.7 Training

The training process is split into multiple steps:

1. The first step is called pretraining, using backpropagation. To the DNN described Sec. 4.3.3, a dropout with a rate of 0.44 and a fully connected layer are added. That new final layer is used to classify the representations, as is usually done for classification tasks. The fully connected layer takes as input a tensor of size p and returns a tensor of size $N - k$. This step is used to initialize the DNN's weights. The reason is that a DNN classifier uses the last layer to determine the probability that the last hidden activation belongs to one class or another. This activation needs to be quite different for samples of different classes. This activation is what we use as representation in the latent space. For this step, a maximum number of 100 epochs is used. Early stopping is implemented, as the training stops after 20 epochs without improvement. The ADAM optimizer is used alongside a cross-entropy loss. Training samples are drawn randomly and grouped

by batch of 64. The learning rate starts at 0.01 and is divided by 3 after 10 epochs without improvement.

2. The second step is the main training phase, where backpropagation is also used. The dropout and fully connected layer added during step 1 are removed. The same learning rate, optimizer, and early stopping are applied. During this phase, one of the two losses, the triplet loss or the prototypical loss, is used. One of the two distances is also used, and will be in the NCM algorithm.

3. Then, a first quantization step is applied. To all layers, a PACT quantizer is used to quantize the input (except the first layer) and output. In our case, we do not use a PACT for the first convolutional layer as the input is the infrared image itself, which is already integers between 0 and 255. Then, as all our layers end with a ReLU activation, the minimum value an input can assume is 0, so m is set to 0. The s and M parameters introduced in Sec. 4.3.6 are trainable. PACT is a linear quantizer. A PACT quantizer is also applied to the output of each layer, following the same algorithm. Because of the integerization of the output, performance may be worse. The weights are quantized using a MaxMinWeight quantizer and the biases with a BiasQuantizer, explained above.

4. After having partly quantized the model, we need to train it with backpropagation one more time. The exact same training algorithm is applied as step 2. When a model is quantized, the weights are less precise and approximations are made during the forward pass, which might lead to poorer performance. The QAT is used to mitigate the performance loss by learning proper PACT quantization parameters and a representation of weights robust to quantization.

5. Our DNN is now fully trained; the weights will not be trained anymore. We can fully integerize our model to let it be exactly the same algorithm it will be once run on the MCU. There are two main differences with the MPS model. First, the PACT quantifiers do not clamp anymore, as we make sure the inputs are in the range $[0, 255]$, they are indeed unsigned integers represented with 8 bits. If we simplify the forward pass of a quantized model by removing the clipping, we obtain the following equation:

$$s_y(\tilde{Y}) = s_x s_w(\tilde{W}\tilde{X} + \tilde{B})$$

which can be rewritten as

$$\tilde{Y} = \frac{s_x s_w}{s_y}(\tilde{W}\tilde{X} + \tilde{B})$$

To avoid computing $\frac{s_x s_w}{s_y}$, and also because we need to have an integer representation of them, we replace this term by an integer scale S and a shift n, such that $\frac{s_x s_w}{s_y}$ is as close as possible to $\frac{S}{2^n}$, using a binary search. Dividing by 2^n is equivalent to right-shifting the bits of a number by n positions, which is an operation that the MCU can do efficiently.

6. Once the DNN is quantized, we use it to compute the $N - k$ prototypes of the classes seen during training. These prototypes will be directly used on the device to classify inputs in the corresponding classes. The k remaining prototypes can be computed directly on the device.

7. Finally, we export the model to C files. Several template files can be completed by extracting the weights, scales, prototypes, biases, etc. from the DNN trained and quantized with Python. The produced code, written in C, can read data from the IR array and classify it, using the NCM algorithm. The code is compiled using riscv32-unknown-elf-gcc [103], a cross-compiler targeting MCU with a RISC-V architecture and without an operating system.

4.3.8 Testing

At the end of each training step, the performance of the algorithm is tested over the evaluation dataset. In each test, we create an empty NCM and use all the training data from the $N - k$ classes to compute a prototype for each class. In a FSL setup, we use a fixed number of instances from each class; otherwise, we use all the training dataset to compute the prototypes. Then, using these prototypes, we compute the accuracy of the algorithm. We do the same operation but using all N classes, in order to compare the performance of the model on seen and unseen data.

In reality, two codes are exported. We use the Spike simulator [104] on one code to simulate it running on a RISC-V architecture. This code checks correctness by computing the accuracy of the exported network before and after training the k remaining prototypes, with 100 samples exported from the training dataset. The other code is meant to run on the MAUPITI device.

It includes reading the data from the sensor, classifying it, and training the NCM. A failure in the compilation of the code indicates too high memory requirements.

Then, the size of the DNN is measured by computing the necessary memory to store the weights, activations, prototype and other variables necessary for the classification algorithm. The size of instructions required to execute the DNN is estimated by using the *size* command on the compiled code, to which the same value obtained from the compilation of a code similar but in which the DNN is not used is subtracted.

Finally, the latency of the algorithm is measured. To do so, the algorithm is executed on the device, but by computing 20 and 100 times the DNN's forward pass and classification through the NCM. The time required to do it, plus getting the values from the sensor, is measured. The values are then used to compute the duration of one single classification.

4.3.9 Implementation

The embedded algorithm executed on the MCU already has $N-k$ prototypes computed, with the memory necessary to compute the k remaining already allocated. The number of samples used to compute a prototype is also stored. When a new sample is to be classified, it is stored in a buffer array, and another empty array is used. They will store alternately the activations. The encoding of the prototype is then returned (it is, in fact, stored in one of the two buffers). This encoding is classified by comparing the distances between it and the computed prototypes. During testing, the output class and the IR image are returned to the controlling computer for visualization. In a real-life application, this class will be sent to a control unit to interact with another system.

To perform the training, a real system would have one or several buttons to specify which class is created or updated, and when. In our system, such a button is not available, so the first 32 samples are used to compute one of the k remaining prototypes. The encodings of these samples are summed and saved in a temporary buffer, component-wise. The buffer is then divided by 32 (which corresponds to a right shift by 5 increments) and saved in the pre-allocated memory.

4.4 Baseline

4.4.1 Tradtional classifier

The algorithm described above is meant to perform classifications of an IR on 5 classes. This task can be easily tackled with a CNN whose last layer is used to compute probabilities. The CNN used to compare its performance with our NCM is the one used in the pretraining phase, so with the same hyperparameters and loss as described in Sec. 4.3. Both algorithms will be trained on all 5 classes of the training dataset and their accuracies over a validation dataset will be measured and analyzed. The accuracy of the NCM will be measured before the quantization, after the finetuning step.

4.4.2 Replay

Modifying a traditional classifier to take into account new classes is not an easy task, as it requires adding a new neuron to the last layer. Then, the DNN needs to be trained with new samples [60]. This algorithm can be simulated offline by updating the DNN with random data from balanced classes. However, this algorithm cannot be executed online, as the memory requirements exceed the available memory. This replay training will be tested offline in a FSL setup to simulate data scarcity. 5 techniques will be tried, and compared with the NCM.

The comparison between the two algorithms will be done for multiple numbers of shots, which is the number of samples for the new class (and the old classes) used to train the DNN to recognize the new class.

The 5 techniques are:

1. Retraining all layers of the DNN
2. Retraining only the last layer without replay
3. Retraining only the last layer of the DNN
4. Retraining only the bias of the last layer
5. Retraining only the new neuron

Chapter 5

Results and discussion

5.1 Training Hyperparameter Selection

The choice of the best model will be made by comparing the performances of the algorithm trained with different hyperparameters, on the following class exclusion setups:

- Class n°1 (empty) is not seen during backpropagation training, the other 4 classes are.
- Class n°4 (right arm raised) is not seen during training, the other 4 classes are.
- Classes n°3 and n°4 are not visible during training; the other three classes are.

In all cases, the (average) F1 score of the excluded class(es), and the accuracy across all classes will be considered. The experiment will be repeated several times to mitigate the effect of randomness (when sampling batches and at initialization). The range of accuracy obtained with the best and worst performers is measured.

The hyperparameters involved in the training of the DNN are summarized Tab. 5.1.

Choosing the correct loss, margin, and distance is important to ensure the NCM is as accurate as possible. The choice of prototype size will also influence performance, but is constrained by memory requirements.

Hyperparameter	Value	Analysis
Learning Rate (LR)	1e-2, divided by 3 after 10 epochs without improvement	
Early-stopping	20 epochs without improvement	
Number of epochs	max. 100	
Training	Phases might be skipped	Sec. 5.6
Batch size	64	
Support & query samples	10 & 30	
Loss	Triplet or Prototypical	Sec. 5.1.1
Distance	Euclidean or Angular	Sec. 5.1.1
Margin	To be tuned	Sec. 5.1.1
Model architecture	Described in 4.3.3	
Prototype size	To be tuned	Sec. 5.1.2

Table 5.1: Hyperparameters

5.1.1 Loss, Margin and Distance

The selection of the loss, margin and distance has been carried out by computing the F1-score of the class not seen during training, if one, or the average of the F1-score of the classes not seen during training, if two. The experiment was performed five times, using five different seeds to assess robustness against randomness. The pretraining step has been skipped in all experiments, and a prototype size of 64 has been selected.

The following combinations of loss, margin, and distances, defined in Sec. 4.3.2 and Sec. 4.3.5, are tested:

- **ATL25** is a triplet loss with a margin of 0.25 and an angular distance metric.
- **ATL50** is a triplet loss with a margin of 0.5 and an angular distance metric.
- **APL0** is a prototypical loss with a margin of 0 and an angular distance metric.
- **APL50** is a prototypical loss with a margin of 0.5 and an angular distance metric.
- **EPL** is a prototypical loss with an Euclidean distance metric.

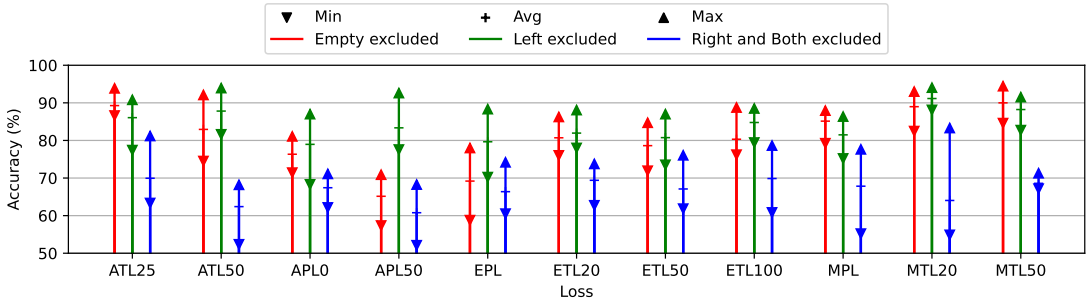
- **ETL20** is a triplet loss with a margin of 20 and a Euclidean distance metric.
- **ETL50** is a triplet loss with a margin of 50 and a Euclidean distance metric.
- **ETL100** is a triplet loss with a margin of 100 and a Euclidean distance metric.
- **MPL** is a prototypical loss with a Manhattan distance metric.
- **MTL20** is a triplet loss with a margin of 20 and a Manhattan distance metric.
- **MTL50** is a triplet loss with a margin of 50 and a Manhattan distance metric.

The results of the experiment are shown Fig. 5.1. By comparing the results on the 3 different setups, it is clear that the first one ("empty" class excluded during training) is the easiest. The empty class is very different from the other classes, so it can naturally be separated without being seen previously. The second setup shows lower accuracy and F1-score, as separating the "left arm" class from the other, similar classes is harder without the DNN being trained on. The setup with two excluded classes is even more challenging.

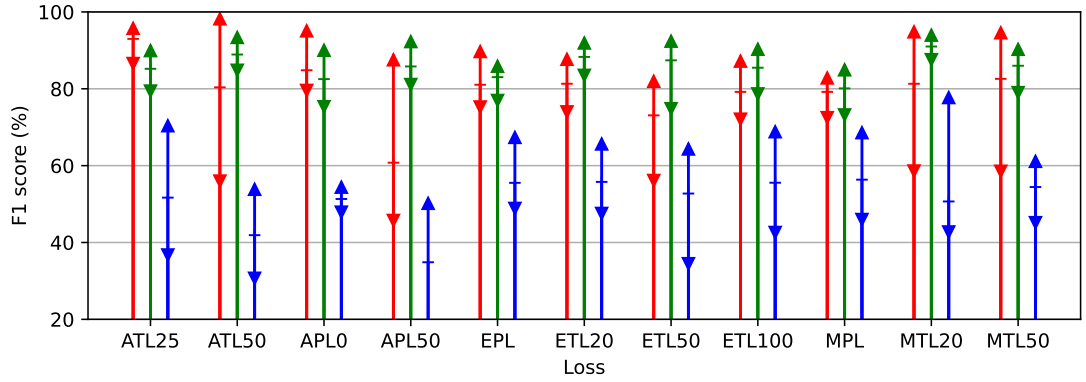
Fig. 5.1a and Fig. 5.1b show that the angular distance is more suited for the task. The ATL25 achieved the best accuracy in all three setups: 89.3%, 86.2%, and 69.9%, demonstrating its ability to discriminate between classes. The APL0, while getting lower accuracy, shows strong and consistent F1 scores for the new class(es), meaning it can classify correctly classes it has not been trained on. However, the angular distance requires more computation when training, as it requires computing a norm, with a square root. The Manhattan distance with the Triplet Loss shows similar, but not robust, performances. For these reasons, the ETL20, which shows the robustest accuracies and F1-scores, even if less accurate, is selected.

5.1.2 Prototype Size

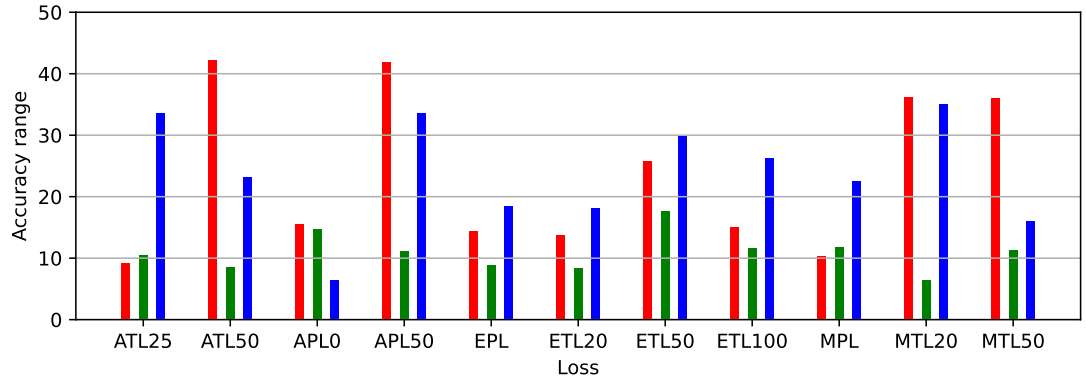
The size of the prototype is very important, as it will influence the performance of the algorithm and the memory and latency requirements. The



(a) Accuracy across all classes



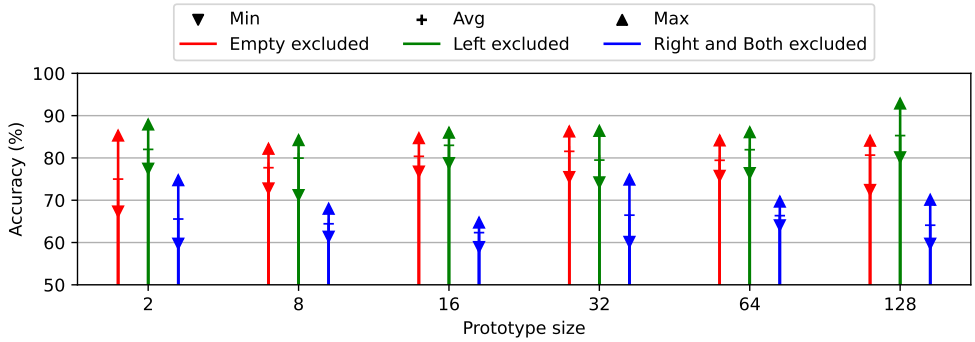
(b) F1 score of the excluded class(es)



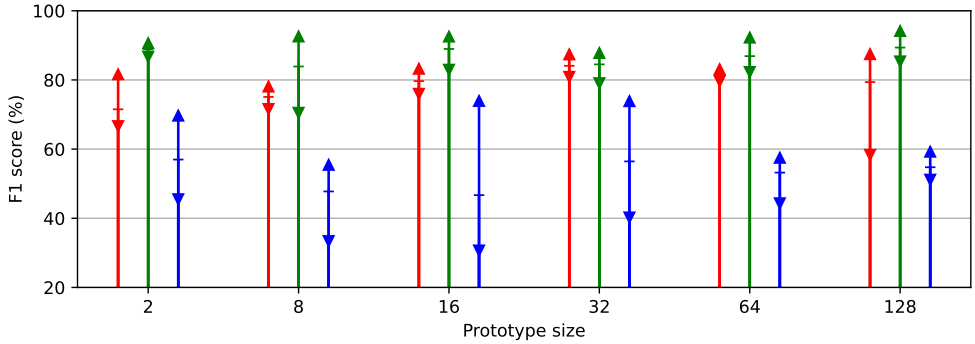
(c) Robustness: Range of the accuracy across all classes

Figure 5.1: Performance of different losses on 3 experiments

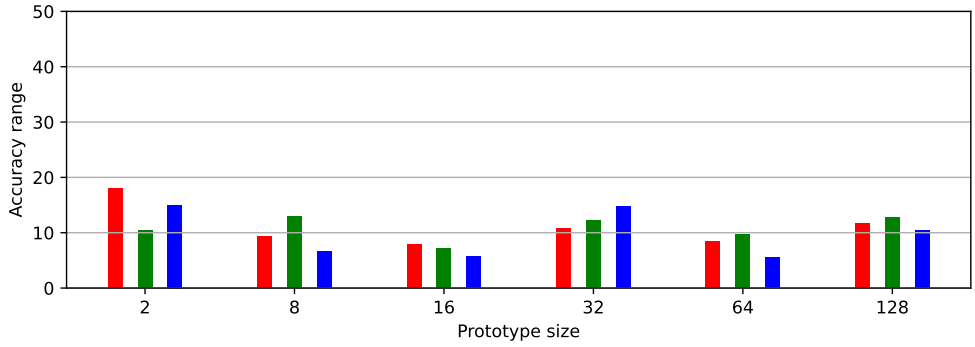
same experiments are performed with the ETL20 loss and various prototype sizes, from 2 to 128. The result is shown on Fig. 5.2.



(a) Accuracy across all classes



(b) F1 score of the excluded class(es)



(c) Robustness: Range of the accuracy across all classes

Figure 5.2: Performance of different prototype sizes on 3 experiments

As Fig. 5.2 shows, the size of the prototypes generally increases the accuracy and F1 score when one new class is added, from 75% to 80.6% for the "empty" class, and from 82.0% to 85.3% for the "right arm" class. For

the third setup, the prototype size of 32 shows the best results, with 66% of accuracy and 56.4% of F1-score. Unlike the various losses, all prototype sizes lead to low accuracy ranges, as shown in Fig. 5.2c, which means the loss choice is the factor influencing the robustness. If we consider the average of the three setups as our selection criterion, then the prototype sizes of 64 and 32 are very close, with a difference of 0.1% on accuracy and 1.5% on F1-score. However, using a prototype size of 64 leads to more robustness.

Finally, a DNN trained with the Euclidean Triplet Loss, with a margin of 20 and a prototype size of 64, is the best algorithm.

5.2 Influence of the datasets on the performance

As explained in Sec. 4.1, both datasets have a slightly different distribution, which is realistic since training an algorithm offline typically uses data from a non-identical environment. Therefore, the algorithm needs to be accurate even when presented with data from a different distribution. On Fig. 5.3, the distribution of each component of the encodings obtained by a NCM with a prototype size of 4 is shown. As explained, both distributions of training and validation samples differ for most classes.

This distribution shift can lead to confusion among classes. Fig. 5.4a is the confusion matrix computed with a NCM trained on all classes, clearly showing that samples from the "right arm" class are regularly classified as "both arms", and that samples from the "both arms" and the "left arm" classes may be classified in the standing class. When the samples from both datasets are mixed to obtain two validation and training datasets with identical distribution, then the confusion matrix, shown in Fig. 5.4b, shows a nearly perfect classification, with an accuracy of 98,73%.

Among the settings of the environment that influence the data distribution, temperature plays a strong role, as 5.5 shows three images recorded in places with 3 different temperatures. The sensor has been calibrated to give accurate results at 20°C. If too cold, the target human is hard to see, and the image is much darker. The arms cannot be perceived. At hotter temperatures, the whole image tends to be red, and the target is again hard to see.

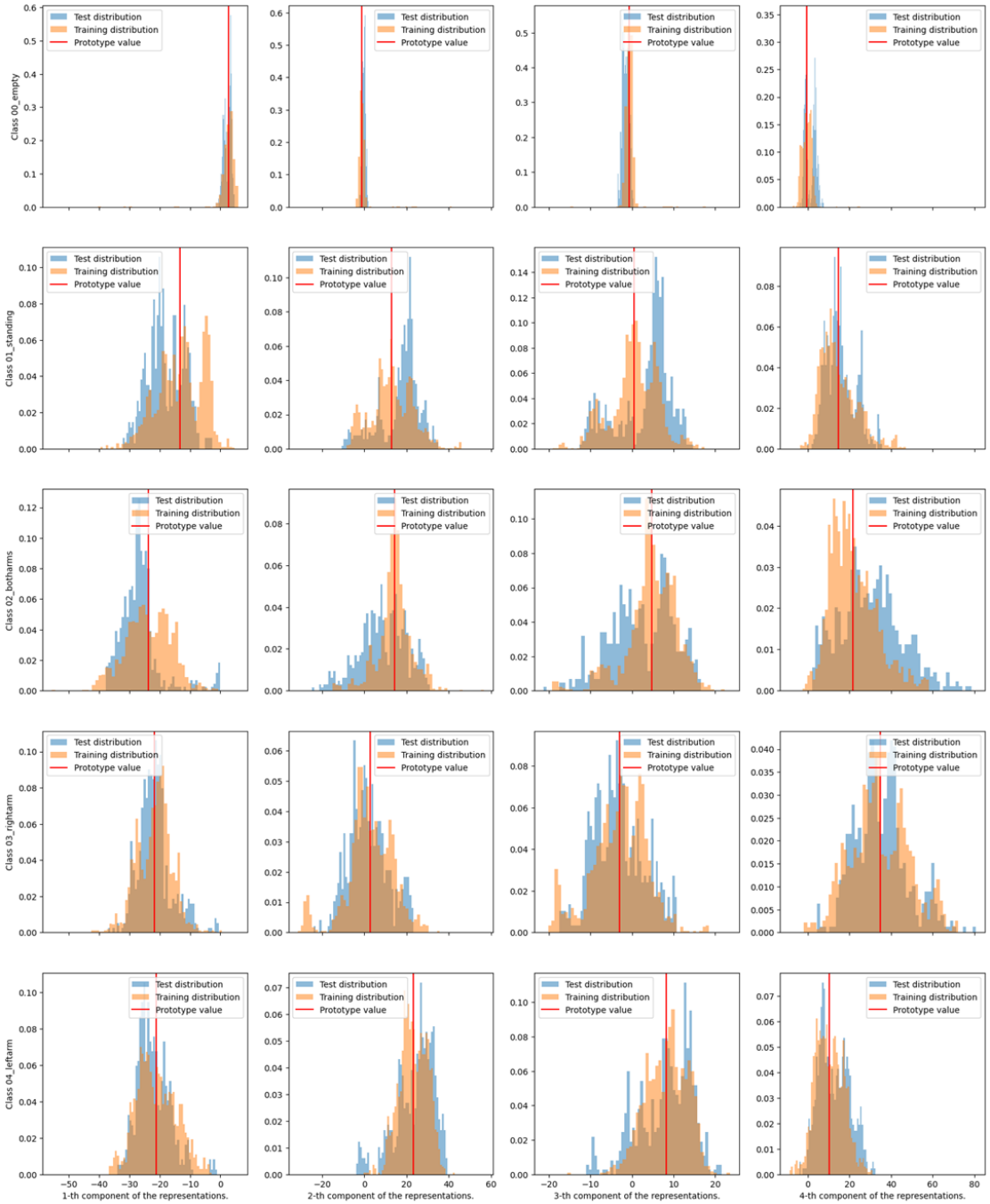
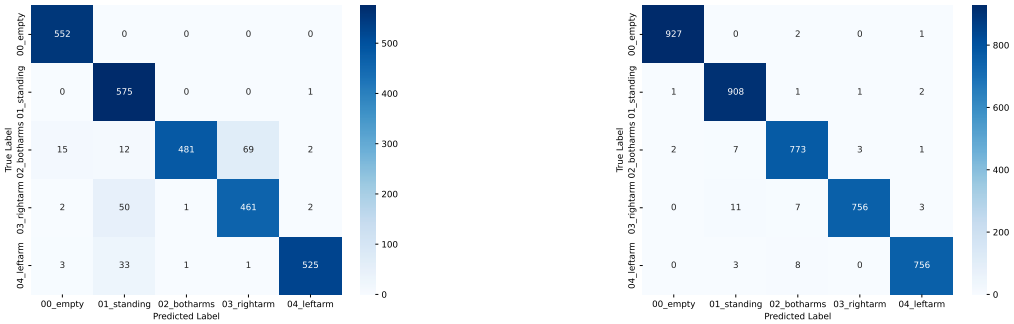
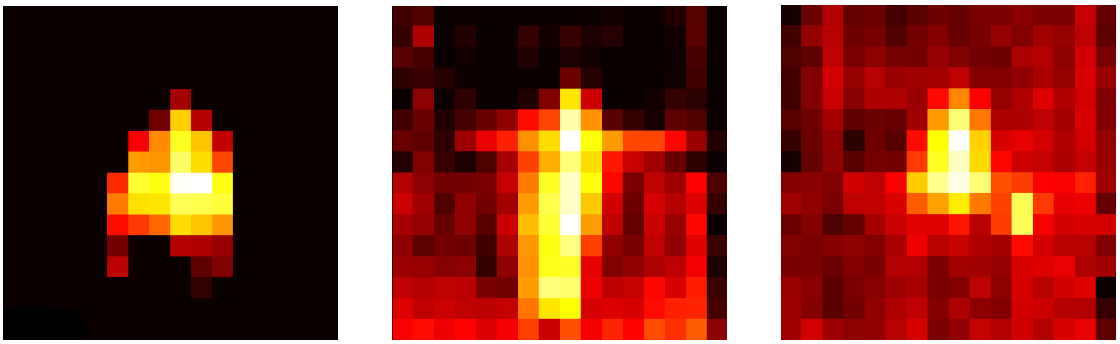


Figure 5.3: Distribution of encodings of both datasets



(a) Separate datasets **(b)** Mixed datasets

Figure 5.4: Confusion matrices with datasets mixed and separated



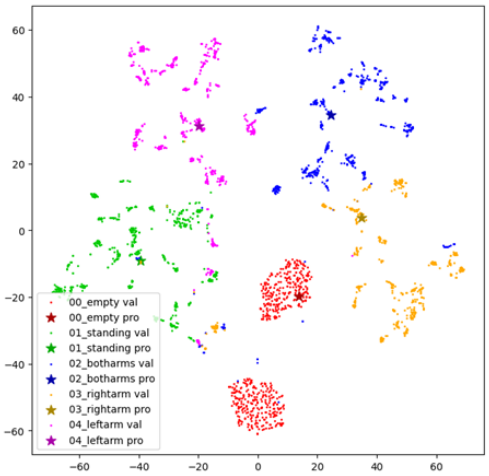
(a) IR image recorded at 17°C **(b)** IR image recorded at 20°C **(c)** IR image recorded at 24°C

Figure 5.5: Influence of the temperature on the recorded image

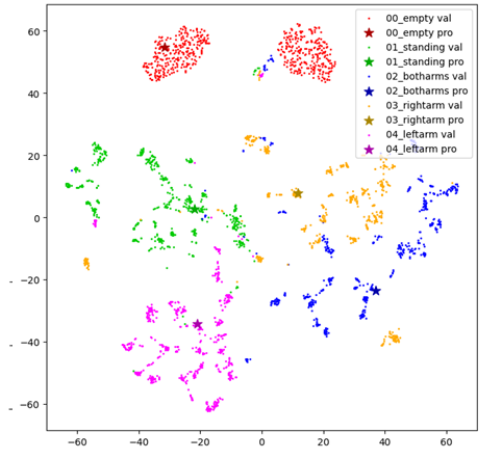
5.3 Assessing the Quality of the Separation

The confusion of the NCM when classifying similar inputs of different classes can be visualized using the t-SNE algorithm [105]. Fig. 5.6 shows a bi-dimensional representation of the encodings of the validation dataset and the prototypes obtained by a non-quantized NCM trained with pretraining and finetuning with the Euclidean Triplet Loss and a margin of 50, on two setups. On the "empty" excluded setup, the four other classes are well separated, leading to very little confusion. The empty class is separated too. Because this class is much different than the other, it is not necessary to have it during training. However, Fig. 5.6b shows that not using a class similar to

other classes during training will lead to a poor separation of this class.



(a) "Empty" class removed during training



(b) "Right arm" class removed during training

Figure 5.6: Visualization of the encodings and prototypes using t-SNE

5.4 Memory Requirements

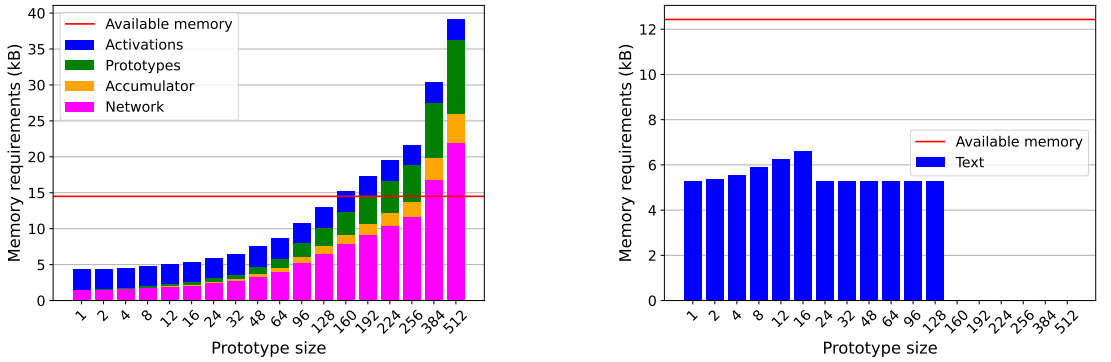
5.4.1 Regarding the Prototype Size

For a given distance, the only parameter that impacts the memory footprint of the algorithm is the prototype size. The use of Euclidean or angular distance can slightly influence the number of instructions, but this difference has not been measured, as the angular distance has not been implemented. The memory requirements have been plotted on Fig. 5.7 for multiple prototype sizes p. The memory footprint can be divided into 5 parts:

- **The activations** correspond to the two buffers used to store alternately the inputs and outputs of each layer. They always have the same size, as the biggest activations are the first two, and the prototype size does not influence the size of the activation buffers.

- Each **prototype** requires p values, stored on 4 bytes, which is the format used for the output of a layer if it is not requantized. For the last layer, it is not necessary to requantize the output.

- **The accumulator** requires $8p$ bytes and is used to update a prototype.
- **The DNN**'s memory footprint corresponds to the weights and biases, plus the scale factors used for quantization. The last layer's number of parameters is proportional to p.
- **The text** footprint corresponds to the compiled instructions, which are stored in a different memory than the data on the considered platform.



(a) Data memory requirements **(b)** Instruction memory requirements

Figure 5.7: Memory requirements of the algorithm

As shown on Fig. 5.7, the critical part is the data memory, as the text memory does not go higher than 7 kB. For $p > 128$, the required memory to store the data exceeds the available 14.5kB, limiting the prototype size to 128. The required memory is an affine function of p, as everything but the size of activations and of the first layers is proportional to p. The shape of the text memory requirement can be explained by inline and non-inline functions, which is a process the compiler can use to optimize code execution. Instead of writing a loop of instructions, it writes the same instruction multiple times. Beyond $p = 16$, an inline function is likely written as a loop instead of as a block of repeated instructions.

For small prototype sizes, it is clear that a more complex DNN, or images with better resolution, could be used to potentially produce more accurate results without exceeding the memory availability. This possibility will be explored in future works.

5.4.2 Regarding the Distance

As explained in Sec. 4.3.2, the implementations of different distances require different variables to be stored. Tab. 5.2 shows the data memory overhead necessary when using a distance different than the Euclidean one, with respect to the number of classes N and the prototype size p.

Distance Name	Memory Overhead (byte)
Manhattan	0
Angular 1	$4N$
Angular 2	$4pN$

Table 5.2: Memory overhead for other distances

The Manhattan distance only requires the prototypes to be saved, like the Euclidean distance, when the first angular implementation requires the prototypes' norms to be saved, and the second implementation requires pre-computed intermediary prototypes to be stored, which have the same memory footprint as the prototypes themselves.

5.5 Latency

5.5.1 Regarding the Prototype Size

The latency of the algorithm has been measured for several prototype sizes and for the traditional classifier.

As Fig. 5.8 shows, the latency increases with the prototype size, with an approximate rate of 0,023 ms per prototype size increment. Most of the time is spent in the first layers of the DNN. The classification algorithms, whether it is the NCM or the traditional one, have very similar latency, below 100ms. As the sensor has a frequency of 10 measures per second, the NCM can be used.

The latency of the training algorithm has been measured too, with a latency overhead below 1 ms. In reality, to compute a new prototype with N samples, then only $(N + 1)p$ operations are necessary on top of the DNN. If backpropagation were used, the latency would go higher than 100 ms, as the backward pass is usually twice to three times longer than the forward pass [106].

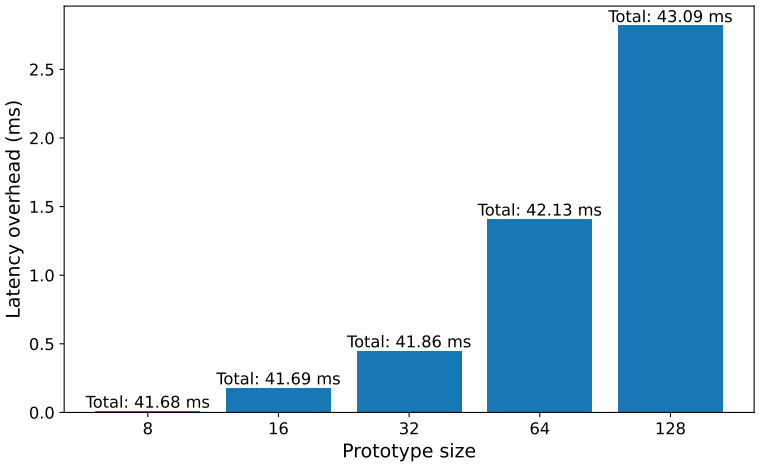


Figure 5.8: Latency of the algorithms

5.5.2 Regarding the Distance

The latency of the other distances has been measured for 5 classes and two prototype sizes: 8 and 32. A comparison is shown Tab. 5.3. As expected, when looking at the distance equations in Sec. 4.3.2, the latencies of the Euclidean, the Manhattan and the second implementation of the angular distances are almost equal, the Manhattan one being slightly faster. The latency of the first angular implementation is only 50 to 70 µs higher than the second implementation, so approximately 10 to 12 µs per class.

Distance	Latency when $p = 8$ (ms)	Latency when $p = 32$
Euclidean	41,62	42,22
Angular 1	41,69	42,27
Angular 2	41,62	42,21
Manhattan	41,62	42,18

Table 5.3: Comparison of latencies of the classifying algorithm for multiple distances and implementations

5.6 Assessing the Relevance of All Training Steps

As mentioned in Sec. 4.3.7, 3 training phases consist of updating the weights of the DNN with the backpropagation and gradient descent algorithm. The pretraining step utilizes a final linear classifier, while the fine-tuning step employs the triplet loss to separate inputs of different classes. Finally, the QAT step does the same, but after quantization. In this section, we assess the utility of each step by comparing the accuracy achieved when skipping one or more of them. The DNN is trained on the classes n°2, n°3, and n°4. The accuracy of the NCM is measured on the 3 classes and on all classes, including the classes n°1 and n°5 that are not seen during training. The Triplet Loss is used with the Euclidean distance and a margin of 50. The prototypes have a size of 64. Every other step is realized, so the DNN is perfectly quantized and can be executed on the device. Each combination of skipped phases is performed nine times to mitigate the effect of random initialization and batches. The maximum, minimum, and average accuracy obtained is shown Fig. 5.9.

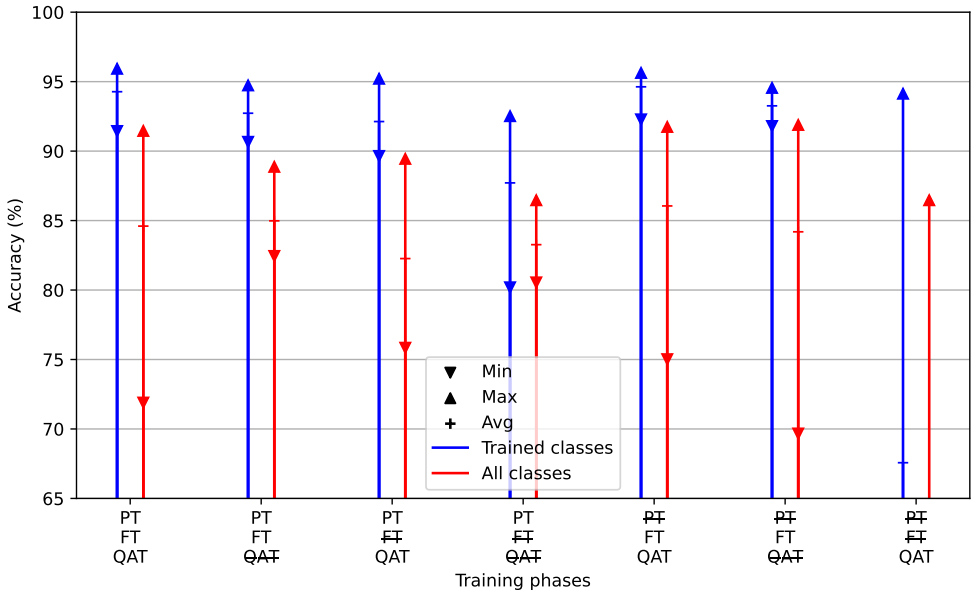


Figure 5.9: Performance of NCMs trained by skipping one or several phases

Fig. 5.9 shows that the QAT step is essential to improve the performance of

the DNN, however, it is not enough, as performing QAT on an untrained DNN is not really efficient. The pretraining step, while expected to help initialize the weights and biases of the CNN to produce class-separated prototypes, does not actually provide any improvement in the NCM's performance, the performance being very similar. It even makes it less robust. Using only pretraining is insufficient to produce a highly performant DNN, as fine-tuning with the triplet loss yields better results.

5.7 Few Shot Learning

To verify if the algorithm can learn new classes with only a few samples, the NCM's prototypes are trained with different numbers of samples from each class. The accuracy obtained by training on fewer samples should be lower than when using all the training samples; however, this accuracy is expected to reach its maximum above a threshold we are determining. We measure accuracy obtained by computing prototypes from the fully integerized DNN, using a balanced subset of data from each class. The network is trained with the triplet loss and the Euclidean distance, with a margin of 50. The prototype size is still set to 64. 3 exclusion setups are considered: Excluding the class n°5, the classes n°1 and n°5, and finally excluding no classes. The accuracy is measured across all classes, even the excluded ones. The pretraining step is skipped.

Figure 5.10 shows the accuracy reached by the algorithm in a FSL setup, for multiple numbers of shots and in 3 exclusion setups. In all 3 cases, the accuracy reached in a 1-shot learning setup is lower by approximately 10%, compared to when all samples are used. In the simplest setup, where all classes are used during training, two shots are sufficient to achieve an accuracy comparable to that obtained with all samples. This result is also very consistent, showing the NCM ability to perform with very few shots on trained classes. In the more difficult setups, the performance grows until reaching a plateau, 16 to 32 shots being enough to obtain the same results as with all samples.

5.8 Comparison with Baseline

The algorithm we explained and analyzed is different than the traditional CNN-based algorithm used for classification tasks. When the traditional

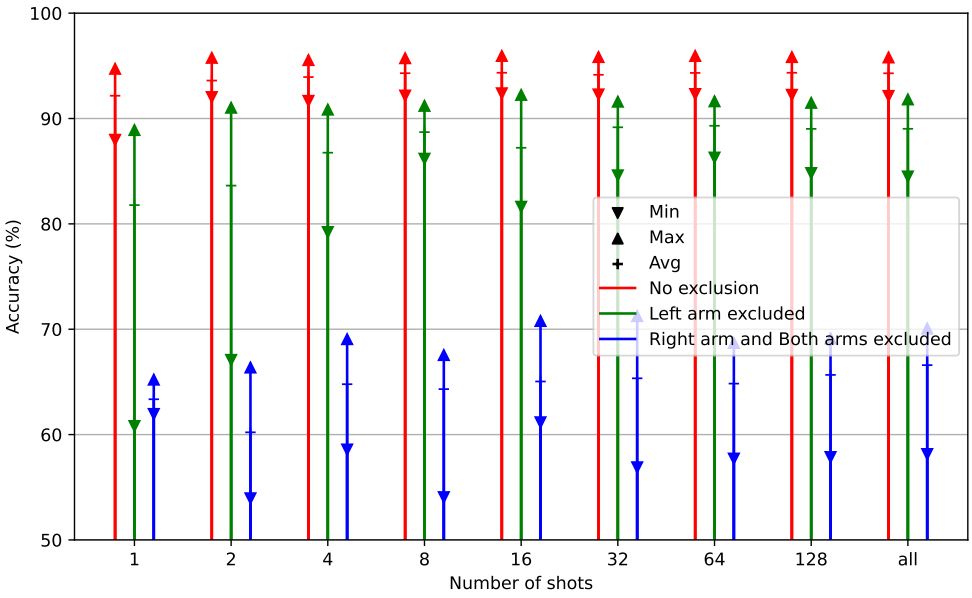


Figure 5.10: Accuracy in FSL setup

algorithm uses a last layer to compute probabilities, we compute distances to average encodings. This traditional algorithm is used as a baseline, as we want to ensure our algorithm has performance on par with the traditional one.

5.8.1 Performance on Classification

First, we want our NCM to be as accurate as the traditional CNN-based classifier when trained and tested on all classes. To verify this, we trained four different traditional classifiers, with the size of the input of the last layer being 8, 16, 32, or 64. We also trained four NCMs with these prototype sizes. As Tab. 5.4 shows, both algorithms reach similar performance, between 93.6% and 95.6%, ensuring that using a NCM does not lead to lower performance.

5.8.2 Continual Learning

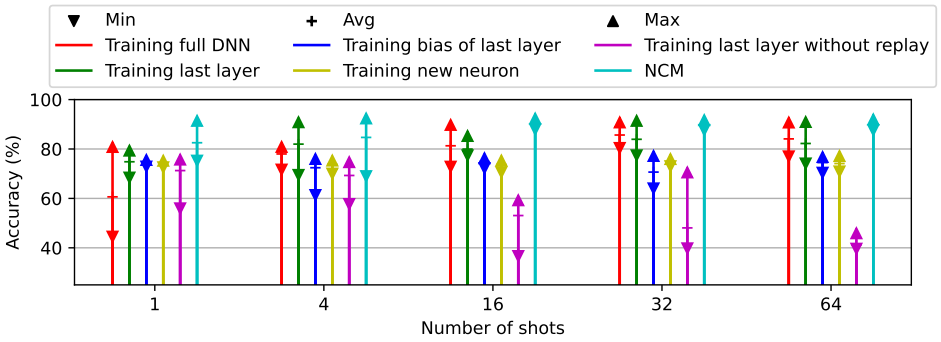
The algorithm used to train a new class when using a traditional classifier seems simple, as it involves adding a new neuron to the last layer and performing a few training epochs with data from this class. In fact, using only data from this class can lead to catastrophic forgetting [59] and requires

Algorithm	Accuracy
NCM $p = 8$	93.645%
NCM $p = 16$	94.330%
NCM $p = 32$	95.625%
NCM $p = 64$	95.07%
Baseline $p = 8$	94.745%
Baseline $p = 16$	94.58%
Baseline $p = 32$	95.315%
Baseline $p = 64$	94.99%

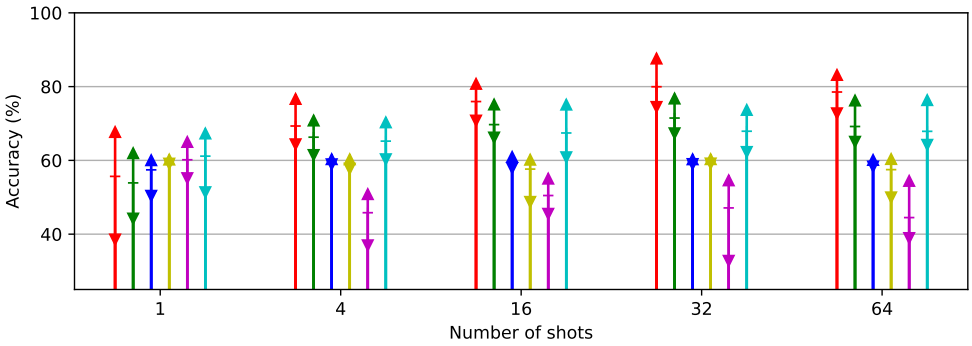
Table 5.4: Comparison of the accuracy of the baseline and NCM algorithm

backpropagation and gradient descent, which cannot be executed on the device, but can be executed on a computer, using Python. This method can be applied to part of or to all of the DNN, using or not replay samples [60]. When using replay, samples from old classes are randomly chosen from the training dataset. First, the left arm is not seen during the first training step before being added, and then the right arm and the left arm classes are not seen during the first training step before being added. The performance of each of these algorithms and the NCM is shown on Fig. 5.11.

As we can see, the best algorithm in the case where two new classes are created is training the full network with replay. When two classes are added, this method is better than the other gradient descent-based methods, but shows an accuracy 5 to 20% lower than the NCM's. Anyway, it cannot be implemented as the memory overhead of the backpropagation is too high. Training the last layer only can be more easily applied to severely constrained devices. The feasibility of this approach on our target device will be assessed in future work. This algorithm gets results similar to the NCM; however, this accuracy is reached after 20 to 30 epochs of training, which can be very long and energy hungry on slow devices. This could be accelerated with a better sampling algorithm for selecting the replay samples [60]. All other algorithms are less accurate than the NCM in both cases.



(a) One class added



(b) Two classes added

Figure 5.11: Accuracy of the baseline and NCM algorithms when adding one or multiple new classes

Chapter 6

Conclusion

This thesis focused on implementing a training algorithm for a DNN algorithm to be executed on a severely constrained device. A NCM algorithm whose DNN is trained using the triplet loss can be trained on a MCU with as little as 32kB of memory. A thermal IR sensor collects an array of IR images that are given to the DNN, whose output is used for classification, being compared to the average output of each class. This average output, called a prototype, can be trained and updated online, without forgetting the previously known classes, storing replay data, or exceeding the memory availability.

The algorithm, when tested on training samples recorded from a real-life environment, can reach up to 95% accuracy, when tested on all the classes used to train the DNN. When tested on unseen classes, it may exceed 90% on some specific setups, but consistently reaches more than 80% on difficult setups. It struggles more on very difficult setups, when multiple, similar classes are not seen during the training of the DNN. The NCM beats most replay-based partial backpropagation algorithms that could be deployed on constrained devices.

When this algorithm succeeds in training and classifying human poses using a shallow CNN with four layers, this task can be considered very easy, given what current DNN algorithms are capable of. For more complex tasks, particularly those involving sensors with higher resolution, the memory requirement can increase and exceed the available memory.

This work proved that performing on-device continual learning is possible on severely constrained devices, using one specific, simple yet effective algorithm, the NCM. This algorithm falls in the Protonet family, from which similar algorithms can be used to perform similar tasks. Other losses may

be used to train the corresponding CNN too. Training algorithms different than Protonets, such as partial backpropagation, can also be applied on the same device. This thesis compared their performance to the NCM on an unconstrained device, but they may be applied on the MCU.

Finally, this work would benefit from being applied to other applications, using the same or different sensors for other classification tasks.

Bibliography

[1] Song Han, Huizi Mao, and William J. Dally. *Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding*. 2016. arXiv: 1510.00149 [cs.CV]. URL: https://arxiv.org/abs/1510.00149 (cit. on pp. 15, 35).

[2] Afonso Lourenço, João Rodrigo, João Gama, and Goreti Marreiros. *On-device edge learning for IoT data streams: a survey*. 2025. arXiv: 2502.17788 [cs.LG]. URL: https://arxiv.org/abs/2502.17788 (cit. on pp. 15, 36).

[3] Nikolaos Argirusis, Achilleas Achilleos, Niyaz Alizadeh, Christos Argirusis, and Georgia Sourkouni. «IR Sensors, Related Materials, and Applications». In: *Sensors* 25.3 (2025), p. 673 (cit. on p. 15).

[4] Jake Snell, Kevin Swersky, and Richard S. Zemel. *Prototypical Networks for Few-shot Learning*. 2017. arXiv: 1703.05175 [cs.LG]. URL: https://arxiv.org/abs/1703.05175 (cit. on pp. 16, 37).

[5] Tyler L. Hayes and Christopher Kanan. *Online Continual Learning for Embedded Devices*. 2022. arXiv: 2203.10681 [cs.LG]. URL: https://arxiv.org/abs/2203.10681 (cit. on pp. 16, 37, 38, 41).

[6] Antoni Rogalski. «Infrared detectors: an overview». In: *Infrared Physics & Technology* 43.3 (2002), pp. 187–210. ISSN: 1350-4495. DOI: https://doi.org/10.1016/S1350-4495(02)00140-8. URL: https://www.sciencedirect.com/science/article/pii/S1350449502001408 (cit. on p. 17).

[7] Roque Alfredo Osornio-Rios, Jose Alfonso Antonino-Daviu, and Rene de Jesus Romero-Troncoso. «Recent Industrial Applications of Infrared Thermography: A Review». In: *IEEE Transactions on Industrial Informatics* 15.2 (2019), pp. 615–625. DOI: 10.1109/TII.2018.2884738 (cit. on p. 18).

[8] Dan Yang, Bin Xu, Kaiyou Rao, and Weihua Sheng. «Passive Infrared (PIR)-Based Indoor Position Tracking for Smart Homes Using Accessibility Maps and A-Star Algorithm». In: *Sensors* 18.2 (2018). ISSN: 1424-8220. DOI: 10.3390/s18020332. URL: https://www.mdpi.com/1424-8220/18/2/332 (cit. on p. 18).

[9] Reza Shoja Ghiass, Ognjen Arandjelović, Abdelhakim Bendada, and Xavier Maldague. «Infrared face recognition: A comprehensive review of methodologies and databases». In: *Pattern Recognition* 47.9 (2014), pp. 2807–2824. ISSN: 0031-3203. DOI: https://doi.org/10.1016/j.patcog.2014.03.015. URL: https://www.sciencedirect.com/science/article/pii/S0031320314001137 (cit. on p. 18).

[10] Ilde Lorato, Tom Bakkes, Sander Stuijk, Mohammed Meftah, and Gerard de Haan. «Unobtrusive Respiratory Flow Monitoring Using a Thermopile Array: A Feasibility Study». In: *Applied Sciences* 9.12 (2019). ISSN: 2076-3417. DOI: 10.3390/app9122449. URL: https://www.mdpi.com/2076-3417/9/12/2449 (cit. on p. 18).

[11] Kosar Khaksari et al. «Review of the efficacy of infrared thermography for screening infectious diseases with applications to COVID-19». In: *Journal of Medical Imaging* 8.S1 (2021), p. 010901. DOI: 10.1117/1.JMI.8.S1.010901. URL: https://doi.org/10.1117/1.JMI.8.S1.010901 (cit. on p. 18).

[12] Linhan Qiao, Shun Li, Youmin Zhang, and Jun Yan. «Early Wildfire Detection and Distance Estimation Using Aerial Visible-Infrared Images». In: *IEEE Transactions on Industrial Electronics* 71.12 (2024), pp. 16695–16705. DOI: 10.1109/TIE.2024.3387089 (cit. on p. 18).

[13] Feng Ling, Giles M. Foody, Hao Du, Xuan Ban, Xiaodong Li, Yihang Zhang, and Yun Du. «Monitoring Thermal Pollution in Rivers Downstream of Dams with Landsat ETM+ Thermal Infrared Images». In: *Remote Sensing* 9.11 (2017). ISSN: 2072-4292. DOI: 10.3390/rs9111175. URL: https://www.mdpi.com/2072-4292/9/11/1175 (cit. on p. 18).

[14] Patrick Hurney, Peter Waldron, Fearghal Morgan, Edward Jones, and Martin Glavin. «Review of pedestrian detection techniques in automotive far-infrared video». In: *IET Intelligent Transport Systems* 9.8 (2015), pp. 824–832. DOI: https://doi.org/10.1049/iet-its.2014.0236. eprint: https://ietresearch.onlinelibrary.

wiley.com/doi/pdf/10.1049/iet-its.2014.0236. URL: https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/iet-its.2014.0236 (cit. on p. 18).

[15] Frank Rosenblatt. «The Perceptron: A Probabilistic Model for Information Storage and organization in the Brain». In: *Psychological Review* 65.6 (1958), pp. 386–408. DOI: 10.1037/h0042519 (cit. on p. 18).

[16] Olga Russakovsky et al. «ImageNet Large Scale Visual Recognition Challenge». In: *International Journal of Computer Vision* 115.3 (2015), pp. 211–252. ISSN: 1573-1405. DOI: 10.1007/s11263-015-0816-y. URL: https://doi.org/10.1007/s11263-015-0816-y (cit. on pp. 18, 23).

[17] Blake Richards et al. «A deep learning framework for neuroscience». In: *Nature Neuroscience* 22 (2019), pp. 1761–1770. DOI: 10.1038/s41593-019-0520-2 (cit. on p. 19).

[18] Damian Podareanu, Valeriu Codreanu, Sandra Aigner, Caspar van Leeuwen, and Volker Weinberg. «Best practice guide-deep learning». In: *Partnership for Advanced Computing in Europe (PRACE), Tech. Rep* 2 (2019) (cit. on p. 20).

[19] Vladimír Kunc and Jiří Kléma. *Three Decades of Activations: A Comprehensive Survey of 400 Activation Functions for Neural Networks.* 2024. arXiv: 2402.09092 [cs.LG]. URL: https://arxiv.org/abs/2402.09092 (cit. on p. 20).

[20] Sergey Ioffe and Christian Szegedy. «Batch normalization: Accelerating deep network training by reducing internal covariate shift». In: *International conference on machine learning.* pmlr. 2015, pp. 448–456 (cit. on p. 21).

[21] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. «Learning representations by back-propagating errors». In: *Nature* 323 (1986), pp. 533–536. URL: https://api.semanticscholar.org/CorpusID:205001834 (cit. on p. 22).

[22] Juan Terven, Diana-Margarita Cordova-Esparza, Julio-Alejandro Romero-González, Alfonso Ramírez-Pedraza, and E. A. hávez-Urbiola. «A comprehensive survey of loss functions and metrics in deep learning». In: *Artificial Intelligence Review* 58.7 (Apr. 2025), p. 195. ISSN: 1573-7462.

DOI: 10.1007/s10462-025-11198-7. URL: https://doi.org/10.1007/s10462-025-11198-7 (cit. on p. 22).

[23] David Shulman. *Optimization Methods in Deep Learning: A Comprehensive Overview*. 2023. arXiv: 2302.09566 [cs.LG]. URL: https://arxiv.org/abs/2302.09566 (cit. on p. 22).

[24] Rajat Kumar Sinha, Ruchi Pandey, and Rohan Pattnaik. *Deep Learning For Computer Vision Tasks: A Review*. 2018. arXiv: 1804.03928 [cs.CV]. URL: https://arxiv.org/abs/1804.03928 (cit. on p. 22).

[25] Junfeng Gao, Yong Yang, Pan Lin, and Dong Sun Park. «Computer Vision in Healthcare Applications». In: *Journal of Healthcare Engineering* 2018 (2018), p. 5157020. DOI: 10.1155/2018/5157020. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5857319/ (cit. on p. 22).

[26] Bimsara Kanchana, Rojith Peiris, Damitha Perera, Dulani Jayasinghe, and Dharshana Kasthurirathna. «Computer Vision for Autonomous Driving». In: *2021 3rd International Conference on Advancements in Computing (ICAC)*. 2021, pp. 175–180. DOI: 10.1109/ICAC54203.2021.9671099 (cit. on p. 22).

[27] Longfei Zhou, Lin Zhang, and Nicholas Konz. «Computer Vision Techniques in Manufacturing». In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 53.1 (2023), pp. 105–117. DOI: 10.1109/TSMC.2022.3166397 (cit. on p. 22).

[28] Li Deng. «Deep learning: from speech recognition to language and multimodal processing». In: *APSIPA Transactions on Signal and Information Processing* 5 (2016), e1. DOI: 10.1017/ATSIP.2015.22 (cit. on p. 22).

[29] Guansong Pang, Chunhua Shen, Longbing Cao, and Anton Van Den Hengel. «Deep Learning for Anomaly Detection: A Review». In: *ACM Comput. Surv.* 54.2 (2021). ISSN: 0360-0300. DOI: 10.1145/3439950. URL: https://doi.org/10.1145/3439950 (cit. on p. 22).

[30] Pradheepan Raghavan and Neamat El Gayar. «Fraud Detection using Machine Learning and Deep Learning». In: *2019 International Conference on Computational Intelligence and Knowledge Economy (ICCIKE)*. 2019, pp. 334–339. DOI: 10.1109/ICCIKE47802.2019.9004231 (cit. on p. 22).

[31] Mufti Mahmud, Mohammed Shamim Kaiser, Amir Hussain, and Stefano Vassanelli. «Applications of Deep Learning and Reinforcement Learning to Biological Data». In: *IEEE Transactions on Neural Networks and Learning Systems* 29.6 (2018), pp. 2063–2079. DOI: 10.1109/TNNLS.2018.2790388 (cit. on p. 22).

[32] T. D'Alessandro, N. D. Cilia, C. De Stefano, F. Fontanella, M. Molinara, A. Scotto di Freca, and I. Marthot-Santaniello. «A Deep Transfer Learning Approach for Writer Identification in Greek Papyri». In: *J. Comput. Cult. Herit.* 18.3 (2025). ISSN: 1556-4673. DOI: 10.1145/3727263. URL: https://doi.org/10.1145/3727263 (cit. on p. 22).

[33] Ayush Singhal, Pradeep Sinha, and Rakesh Pant. «Use of Deep Learning in Modern Recommendation System: A Summary of Recent Works». In: *International Journal of Computer Applications* 180.7 (2017), pp. 17–22. ISSN: 0975-8887. DOI: 10.5120/ijca2017916055. URL: http://dx.doi.org/10.5120/ijca2017916055 (cit. on p. 22).

[34] Zhe Li, Qian He, and Jingyue Li. «A survey of deep learning-driven architecture for predictive maintenance». In: *Engineering Applications of Artificial Intelligence* 133 (2024), p. 108285. ISSN: 0952-1976. DOI: https://doi.org/10.1016/j.engappai.2024.108285. URL: https://www.sciencedirect.com/science/article/pii/S0952197624004433 (cit. on p. 22).

[35] Fatima Dakalbab, Manar Abu Talib, Qassim Nasir, and Tracy Saroufil. «Artificial intelligence techniques in financial trading: A systematic literature review». In: *Journal of King Saud University - Computer and Information Sciences* 36.3 (2024), p. 102015. ISSN: 1319-1578. DOI: https://doi.org/10.1016/j.jksuci.2024.102015. URL: https://www.sciencedirect.com/science/article/pii/S1319157824001046 (cit. on p. 22).

[36] Haidong Li, Jiongcheng Li, Xiaoming Guan, Binghao Liang, Yuting Lai, and Xinglong Luo. «Research on Overfitting of Deep Learning». In: *2019 15th International Conference on Computational Intelligence and Security (CIS)*. 2019, pp. 78–81. DOI: 10.1109/CIS.2019.00025 (cit. on p. 23).

[37] Ching-Chen Wang, Ching-Te Chiu, and Jheng-Yi Chang. «EfficientNet-eLite: Extremely Lightweight and Efficient CNN Models for Edge Devices by Network Candidate Search». In: *Journal of Signal Processing*

Systems 95 (2022), pp. 1–13. DOI: 10.1007/s11265-022-01808-w (cit. on pp. 24, 35).

[38] Song Han, Jeff Pool, John Tran, and William J. Dally. «Learning both weights and connections for efficient neural networks». In: *Proceedings of the 29th International Conference on Neural Information Processing Systems - Volume 1*. NIPS'15. Montreal, Canada: MIT Press, 2015, pp. 1135–1143 (cit. on p. 25).

[39] Leonardo Cecconi, Sander Smets, Luca Benini, and Marian Verhelst. «Optimal Tiling Strategy for Memory Bandwidth Reduction for CNNs». In: *Advanced Concepts for Intelligent Vision Systems*. Ed. by Jacques Blanc-Talon, Rudi Penne, Wilfried Philips, Dan Popescu, and Paul Scheunders. Cham: Springer International Publishing, 2017, pp. 89–100. ISBN: 978-3-319-70353-4 (cit. on p. 25).

[40] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. «Understanding Reuse, Performance, and Hardware Cost of DNN Dataflow: A Data-Centric Approach». In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-52. Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 754–768. ISBN: 9781450369381. DOI: 10.1145/3352460.3358252. URL: https://doi.org/10.1145/3352460.3358252 (cit. on p. 25).

[41] Goran S. Nikolić, Bojan R. Dimitrijević, Tatjana R. Nikolić, and Mile K. Stojcev. «A Survey of Three Types of Processing Units: CPU, GPU and TPU». In: *2022 57th International Scientific Conference on Information, Communication and Energy Systems and Technologies (ICEST)*. 2022, pp. 1–6. DOI: 10.1109/ICEST55168.2022.9828625 (cit. on p. 25).

[42] Maurizio Capra, Beatrice Bussolino, Alberto Marchisio, Guido Masera, Maurizio Martina, and Muhammad Shafique. «Hardware and Software Optimizations for Accelerating Deep Neural Networks: Survey of Current Trends, Challenges, and the Road Ahead». In: *IEEE Access* 8 (2020), pp. 225134–225180. ISSN: 2169-3536. DOI: 10.1109/access.2020.3039858. URL: http://dx.doi.org/10.1109/ACCESS.2020.3039858 (cit. on p. 25).

[43] Shahanur Alam, Chris Yakopcic, Qing Wu, Mark Barnell, Simon Khan, and Tarek M. Taha. «Survey of Deep Learning Accelerators for Edge and Emerging Computing». In: *Electronics* 13.15 (2024). ISSN: 2079-9292. DOI: 10.3390/electronics13152988. URL: https://www.mdpi.com/2079-9292/13/15/2988 (cit. on p. 25).

[44] Andrew Anderson, Aravind Vasudevan, Cormac Keane, and David Gregg. *Low-memory GEMM-based convolution algorithms for deep neural networks*. 2017. arXiv: 1709.03395 [cs.CV]. URL: https://arxiv.org/abs/1709.03395 (cit. on p. 26).

[45] Haoyu Wang and Chengguang Ma. «An optimization of im2col, an important method of CNNs, based on continuous address access». In: *2021 IEEE International Conference on Consumer Electronics and Computer Engineering (ICCECE)*. 2021, pp. 314–320. DOI: 10.1109/ICCECE51280.2021.9342343 (cit. on p. 26).

[46] Chi-Feng Wang. «A basic introduction to separable convolutions». In: *Towards Data Science* 13 (2018), p. 13 (cit. on p. 26).

[47] Jaekyun Ko, Wanuk Choi, and Sanghwan Lee. «PEIPNet: Parametric Efficient Image-Inpainting Network with Depthwise and Pointwise Convolution». In: *Sensors* 23.19 (2023). ISSN: 1424-8220. DOI: 10.3390/s23198313. URL: https://www.mdpi.com/1424-8220/23/19/8313 (cit. on p. 26).

[48] Pengzhen Ren, Yun Xiao, Xiaojun Chang, Po-yao Huang, Zhihui Li, Xiaojiang Chen, and Xin Wang. «A Comprehensive Survey of Neural Architecture Search: Challenges and Solutions». In: *ACM Comput. Surv.* 54.4 (May 2021). ISSN: 0360-0300. DOI: 10.1145/3447582. URL: https://doi.org/10.1145/3447582 (cit. on pp. 26, 27, 35).

[49] M. Augasta and T. Kathirvalavakumar. In: *Open Computer Science* 3.3 (2013), pp. 105–115. DOI: doi:10.2478/s13537-013-0109-x. URL: https://doi.org/10.2478/s13537-013-0109-x (cit. on pp. 27, 28).

[50] Maying Shen, Hongxu Yin, Pavlo Molchanov, Lei Mao, Jianna Liu, and Jose M. Alvarez. «Structural Pruning via Latency-Saliency Knapsack». In: *Advances in Neural Information Processing Systems*. Ed. by S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh. Vol. 35. Curran Associates, Inc., 2022, pp. 12894–12908. URL: https://proceedings.neurips.cc/paper_files/paper/2022/file/

5434be94e82c54327bb9dcaf7fca52b6-Paper-Conference.pdf (cit. on p. 28).

[51] John Osorio, Adrià Armejach, Eric Petit, Greg Henry, and Marc Casas. «A BF16 FMA is All You Need for DNN Training». In: *IEEE Transactions on Emerging Topics in Computing* 10.3 (2022), pp. 1302–1314. DOI: 10.1109/TETC.2022.3187770 (cit. on p. 28).

[52] Shien Zhu, Luan H. K. Duong, and Weichen Liu. «XOR-Net: An Efficient Computation Pipeline for Binary Neural Network Inference on Edge Devices». In: *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*. 2020, pp. 124–131. DOI: 10.1109/ICPADS51040.2020.00026 (cit. on p. 28).

[53] Xiao Sun et al. «Ultra-Low Precision 4-bit Training of Deep Neural Networks». In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin. Vol. 33. Curran Associates, Inc., 2020, pp. 1796–1807. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/13b919438259814cd5be8cb45877d577-Paper.pdf (cit. on p. 28).

[54] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. «A survey of quantization methods for efficient neural network inference». In: *Low-power computer vision*. Chapman and Hall/CRC, 2022, pp. 291–326 (cit. on pp. 28–30).

[55] Yuhang Li, Xin Dong, and Wei Wang. *Additive Powers-of-Two Quantization: An Efficient Non-uniform Discretization for Neural Networks*. 2020. arXiv: 1909.13144 [cs.LG]. URL: https://arxiv.org/abs/1909.13144 (cit. on p. 29).

[56] Edward H. Lee, Daisuke Miyashita, Elaina Chai, Boris Murmann, and S. Simon Wong. «LogNet: Energy-efficient neural networks using logarithmic computation». In: *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2017, pp. 5900–5904. DOI: 10.1109/ICASSP.2017.7953288 (cit. on p. 29).

[57] Minsik Cho, Keivan A. Vahid, Saurabh Adya, and Mohammad Rastegari. *DKM: Differentiable K-Means Clustering Layer for Neural Network Compression*. 2022. arXiv: 2108.12659 [cs.LG]. URL: https://arxiv.org/abs/2108.12659 (cit. on p. 30).

[58] Anuj Diwan, Ching-Feng Yeh, Wei-Ning Hsu, Paden Tomasello, Eunsol Choi, David Harwath, and Abdelrahman Mohamed. «Continual Learning for On-Device Speech Recognition Using Disentangled Conformers». In: *ICASSP 2023 - 2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, June 2023, pp. 1–5. DOI: 10.1109/icassp49357.2023.10095484. URL: http://dx.doi.org/10.1109/ICASSP49357.2023.10095484 (cit. on p. 32).

[59] Michael McCloskey and Neal J. Cohen. «Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem». In: ed. by Gordon H. Bower. Vol. 24. Psychology of Learning and Motivation. Academic Press, 1989, pp. 109–165. DOI: https://doi.org/10.1016/S0079-7421(08)60536-8. URL: https://www.sciencedirect.com/science/article/pii/S0079742108605368 (cit. on pp. 32, 36, 65).

[60] Tyler L. Hayes, Nathan D. Cahill, and Christopher Kanan. *Memory Efficient Experience Replay for Streaming Learning*. 2019. arXiv: 1809.05922 [cs.LG]. URL: https://arxiv.org/abs/1809.05922 (cit. on pp. 32, 36, 37, 39, 50, 66).

[61] Kan Wu, Jinnian Zhang, Houwen Peng, Mengchen Liu, Bin Xiao, Jianlong Fu, and Lu Yuan. *TinyViT: Fast Pretraining Distillation for Small Vision Transformers*. Ed. by Shai Avidan, Gabriel Brostow, Moustapha Cissé, Giovanni Maria Farinella, and Tal Hassner. Cham, 2022 (cit. on p. 35).

[62] Robert David et al. «TensorFlow Lite Micro: Embedded Machine Learning for TinyML Systems». In: *Proceedings of Machine Learning and Systems*. Ed. by A. Smola, A. Dimakis, and I. Stoica. Vol. 3. 2021, pp. 800–811. URL: https://proceedings.mlsys.org/paper_files/paper/2021/file/6c44dc73014d66ba49b28d483a8f8b0d-Paper.pdf (cit. on p. 35).

[63] PyTorch Team. *ExecuTorch*. https://github.com/pytorch/executorch. PyTorch-based lightweight edge AI runtime. 2025 (cit. on p. 35).

[64] Intel Corporation. *OpenVINO Toolkit*. https://github.com/openvinotoolkit/openvino. Deep learning inference and deployment toolkit for edge AI. 2025 (cit. on p. 35).

[65] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. 2017. arXiv: 1704.04861 [cs.CV]. URL: https://arxiv.org/abs/1704.04861 (cit. on p. 35).

[66] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. «MobileNetV2: Inverted Residuals and Linear Bottlenecks». In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2018 (cit. on p. 35).

[67] Brett Koonce. «MobileNetV3». In: *Convolutional Neural Networks with Swift for Tensorflow: Image Recognition and Dataset Categorization*. Berkeley, CA: Apress, 2021, pp. 125–144. ISBN: 978-1-4842-6168-2. DOI: 10.1007/978-1-4842-6168-2_11. URL: https://doi.org/10.1007/978-1-4842-6168-2_11 (cit. on p. 35).

[68] Danfeng Qin et al. «MobileNetV4: Universal Models for the Mobile Ecosystem». In: *Computer Vision – ECCV 2024*. Ed. by Aleš Leonardis, Elisa Ricci, Stefan Roth, Olga Russakovsky, Torsten Sattler, and Gül Varol. Cham: Springer Nature Switzerland, 2025, pp. 78–96. ISBN: 978-3-031-73661-2 (cit. on p. 35).

[69] Delia Velasco-Montero, J. Fernández-Berni, R. Carmona-Galán, and Ángel Rodríguez-Vázquez. *Performance Analysis of Real-Time DNN Inference on Raspberry Pi*. 2018 (cit. on p. 35).

[70] Francesco Daghero, Alessio Burrello, Chen Xie, Marco Castellano, Luca Gandolfi, Andrea Calimera, Enrico Macii, Massimo Poncino, and Daniele Jahier Pagliari. «Human Activity Recognition on Microcontrollers with Quantized and Adaptive Deep Neural Networks». In: *ACM Transactions on Embedded Computing Systems* 21.4 (July 2022), pp. 1–28. ISSN: 1558-3465. DOI: 10.1145/3542819. URL: http://dx.doi.org/10.1145/3542819 (cit. on p. 35).

[71] Vessela Krasteva, Todor Stoyanov, and Irena Jekova. «Implementing Deep Neural Networks on ARM-Based Microcontrollers: Application for Ventricular Fibrillation Detection». In: *Applied Sciences* 15.4 (2025). ISSN: 2076-3417. DOI: 10.3390/app15041965. URL: https://www.mdpi.com/2076-3417/15/4/1965 (cit. on p. 35).

[72] Camilo A. Ruiz-Beltrán, Adrián Romero-Garcés, Martín González-García, Rebeca Marfil, and Antonio Bandera. «FPGA-Based CNN for Eye Detection in an Iris Recognition at a Distance System». In: *Electronics* 12.22 (2023). ISSN: 2079-9292. DOI: 10.3390/electronics12224713. URL: https://www.mdpi.com/2079-9292/12/22/4713 (cit. on p. 35).

[73] Walther Carballo-Hernández, François Berry, Maxime Pelcat, and Miguel Arias-Estrada. «Towards Embedded Heterogeneous FPGA-GPU Smart Camera Architectures for CNN Inference». In: *Proceedings of the 13th International Conference on Distributed Smart Cameras*. ICDSC 2019. Trento, Italy: Association for Computing Machinery, 2019. ISBN: 9781450371896. DOI: 10.1145/3349801.3357136. URL: https://doi.org/10.1145/3349801.3357136 (cit. on p. 35).

[74] Zahra Solatidehkordi, Jayroop Ramesh, A.R. Al-Ali, Ahmed Osman, and Mostafa Shaaban. «An IoT deep learning-based home appliances management and classification system». In: *Energy Reports* 9 (2023). 2022 The 3rd International Conference on Power, Energy and Electrical Engineering, pp. 503–509. ISSN: 2352-4847. DOI: https://doi.org/10.1016/j.egyr.2023.01.071. URL: https://www.sciencedirect.com/science/article/pii/S2352484723000793 (cit. on p. 36).

[75] Youn Joo Lee, Jun Young Hwang, Jiwon Park, Ho Gi Jung, and Jae Kyu Suhr. «Deep Neural Network-Based Flood Monitoring System Fusing RGB and LWIR Cameras for Embedded IoT Edge Devices». In: *Remote Sensing* 16.13 (2024). ISSN: 2072-4292. DOI: 10.3390/rs16132358. URL: https://www.mdpi.com/2072-4292/16/13/2358 (cit. on p. 36).

[76] Jahmunah Vicnesh, Massimo Salvi, Yuki Hagiwara, Hah Yan Yee, Hasan Mir, Prabal Datta Barua, Subrata Chakraborty, Filippo Molinari, and U. Rajendra Acharya. «Application of Infrared Thermography and Artificial Intelligence in Healthcare: A Systematic Review of Over a Decade (2013–2024)». In: *IEEE Access* 13 (2025), pp. 5949–5973. DOI: 10.1109/ACCESS.2024.3522251 (cit. on p. 36).

[77] Yordanka Karayaneva, Sara Sharifzadeh, Yanguo Jing, and Bo Tan. «Human Activity Recognition for AI-Enabled Healthcare Using Low-Resolution Infrared Sensor Data». In: *Sensors* 23.1 (2023). ISSN: 1424-8220. DOI: 10.3390/s23010478. URL: https://www.mdpi.com/1424-8220/23/1/478 (cit. on p. 36).

[78] P. Mohamed Shakeel, Tarek E. El. Tobely, Haytham Al-Feel, Gunasekaran Manogaran, and S. Baskar. «Neural Network Based Brain Tumor Detection Using Wireless Infrared Imaging Sensor». In: *IEEE Access* 7 (2019), pp. 5577–5588. DOI: 10.1109/ACCESS.2018.2883957 (cit. on p. 36).

[79] Wartini Ng, Budiman Minasny, Maryam Montazerolghaem, Jose Padarian, Richard Ferguson, Scarlett Bailey, and Alex B. McBratney. «Convolutional neural network for simultaneous prediction of several soil properties using visible/near-infrared, mid-infrared, and their combined spectra». In: *Geoderma* 352 (2019), pp. 251–267. ISSN: 0016-7061. DOI: https://doi.org/10.1016/j.geoderma.2019.06.016. URL: https://www.sciencedirect.com/science/article/pii/S0016706119300588 (cit. on p. 36).

[80] Guwon Jung, Son Gyo Jung, and Jacqueline M Cole. «Automatic materials characterization from infrared spectra using convolutional neural networks». In: *Chemical Science* 14.13 (2023), pp. 3600–3609 (cit. on p. 36).

[81] Min Peng, Chongyang Wang, Tong Chen, and Guangyuan Liu. «Nirfacenet: A convolutional neural network for near-infrared face identification». In: *Information* 7.4 (2016), p. 61 (cit. on p. 36).

[82] Yuxia Duan et al. «Automated defect classification in infrared thermography based on a neural network». In: *NDT & E International* 107 (2019), p. 102147. ISSN: 0963-8695. DOI: https://doi.org/10.1016/j.ndteint.2019.102147. URL: https://www.sciencedirect.com/science/article/pii/S0963869518306698 (cit. on p. 36).

[83] Young D. Kwon, Rui Li, Stylianos I. Venieris, Jagmohan Chauhan, Nicholas D. Lane, and Cecilia Mascolo. *TinyTrain: Resource-Aware Task-Adaptive Sparse Training of DNNs at the Data-Scarce Edge*. 2024. arXiv: 2307.09988 [cs.LG]. URL: https://arxiv.org/abs/2307.09988 (cit. on p. 36).

[84] Han Cai, Chuang Gan, Ligeng Zhu, and Song Han. *TinyTL: Reduce Activations, Not Trainable Parameters for Efficient On-Device Learning*. 2021. arXiv: 2007.11622 [cs.CV]. URL: https://arxiv.org/abs/2007.11622 (cit. on p. 36).

[85] Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. *On-Device Training Under 256KB Memory*. 2024. arXiv: 2206.15472 [cs.CV]. URL: https://arxiv.org/abs/2206.15472 (cit. on p. 36).

[86] Qin Wang, Olga Fink, Luc Van Gool, and Dengxin Dai. «Continual Test-Time Domain Adaptation». In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2022, pp. 7201–7211 (cit. on p. 36).

[87] Jacob Menick, Erich Elsen, Utku Evci, Simon Osindero, Karen Simonyan, and Alex Graves. *A Practical Sparse Approximation for Real Time Recurrent Learning*. 2020. arXiv: 2006.07232 [cs.LG]. URL: https://arxiv.org/abs/2006.07232 (cit. on p. 36).

[88] Adam N. McCaughan, Bakhrom G. Oripov, Natesh Ganesh, Sae Woo Nam, Andrew Dienstfrey, and Sonia M. Buckley. «Multiplexed gradient descent: Fast online training of modern datasets on hardware neural networks without backpropagation». In: *APL Machine Learning* 1.2 (June 2023), p. 026118. ISSN: 2770-9019. DOI: 10.1063/5.0157645. eprint: https://pubs.aip.org/aip/aml/article-pdf/doi/10.1063/5.0157645/18017061/026118_1_5.0157645.pdf. URL: https://doi.org/10.1063/5.0157645 (cit. on p. 37).

[89] Yequan Zhao, Hai Li, Ian Young, and Zheng Zhang. «Poor Man's Training on MCUs: A Memory-Efficient Quantized Back-Propagation-Free Approach». In: *ACM Trans. Des. Autom. Electron. Syst.* 30.5 (Aug. 2025). ISSN: 1084-4309. DOI: 10.1145/3745772. URL: https://doi.org/10.1145/3745772 (cit. on p. 37).

[90] Keisuke Sugiura and Hiroki Matsutani. *ElasticZO: A Memory-Efficient On-Device Learning with Combined Zeroth- and First-Order Optimization*. 2025. arXiv: 2501.04287 [cs.LG]. URL: https://arxiv.org/abs/2501.04287 (cit. on p. 37).

[91] Li Yang, Adnan Siraj Rakin, and Deliang Fan. «Rep-Net: Efficient On-Device Learning via Feature Reprogramming». In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2022, pp. 12277–12286 (cit. on p. 37).

[92] Christos Profentzas, Magnus Almgren, and Olaf Landsiedel. «MiniLearn: On-Device Learning for Low-Power IoT Devices.» In: *EWSN*. 2022, pp. 1–11 (cit. on p. 37).

[93] Gunshi Gupta, Karmesh Yadav, and Liam Paull. «Look-ahead Meta Learning for Continual Learning». In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin. Vol. 33. Curran Associates, Inc., 2020, pp. 11588–11598. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/85b9a5ac91cd629bd3afe396ec07270a-Paper.pdf (cit. on p. 37).

[94] Jianheng Huang, Leyang Cui, Ante Wang, Chengyi Yang, Xinting Liao, Linfeng Song, Junfeng Yao, and Jinsong Su. *Mitigating Catastrophic Forgetting in Large Language Models with Self-Synthesized Rehearsal*. 2024. arXiv: 2403.01244 [cs.CL]. URL: https://arxiv.org/abs/2403.01244 (cit. on p. 37).

[95] Tyler L. Hayes, Kushal Kafle, Robik Shrestha, Manoj Acharya, and Christopher Kanan. «REMIND Your Neural Network to Prevent Catastrophic Forgetting». In: *Computer Vision – ECCV 2020*. Ed. by Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm. Cham: Springer International Publishing, 2020, pp. 466–483. ISBN: 978-3-030-58598-3 (cit. on p. 37).

[96] Tyler L. Hayes and Christopher Kanan. «Lifelong Machine Learning With Deep Streaming Linear Discriminant Analysis». In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*. June 2020 (cit. on p. 38).

[97] *lowRISC / Ibex: A 32-bit RISC-V CPU core*. GitHub repository. https://github.com/lowRISC/ibex (cit. on p. 40).

[98] Matteo Risso, Chen Xie, Francesco Daghero, Alessio Burrello, Seyedmorteza Mollaei, Marco Castellano, Enrico Macii, Massimo Poncino, and Daniele Jahier Pagliari. «HW-SW Optimization of DNNs for Privacy-Preserving People Counting on Low-Resolution Infrared Arrays». In: *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2024, pp. 1–6. DOI: 10.23919/DATE58400.2024.10546798 (cit. on p. 40).

[99] Chen Xiea, Francesco Daghero, Yukai Chen, Marco Castellano, Luca Gandolfi, Andrea Calimera, Enrico Macii, Massimo Poncino, and Daniele Jahier Pagliari. «Efficient Deep Learning Models for Privacy-Preserving People Counting on Low-Resolution Infrared Arrays». In:

IEEE Internet of Things Journal 10.15 (2023), pp. 13895–13907. ISSN: 2327-4662. DOI: 10.1109/JIOT.2023.3263290 (cit. on p. 42).

[100] D. Jahier Pagliari, M. Risso, B. A. Motetti, and A. Burrello. *PLiNIO: A User-Friendly Library of Gradient-based Methods for Complexity-aware DNN Optimization*. 2023. arXiv: 2307.09488 [cs.LG] (cit. on p. 43).

[101] Manuele Rusci and Tinne Tuytelaars. *Few-Shot Open-Set Learning for On-Device Customization of KeyWord Spotting Systems*. 2023. arXiv: 2306.02161 [cs.LG]. URL: https://arxiv.org/abs/2306.02161 (cit. on p. 44).

[102] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. *PACT: Parameterized Clipping Activation for Quantized Neural Networks*. 2018. arXiv: 1805.06085 [cs.CV]. URL: https://arxiv.org/abs/1805.06085 (cit. on p. 45).

[103] *RISC-V GNU Compiler Toolchain*. https://github.com/riscv-collab/riscv-gnu-toolchain. Accessed September 2025 (cit. on p. 48).

[104] RISC-V International. *Spike: RISC-V ISA Simulator*. https://github.com/riscv-software-src/riscv-isa-sim. Accessed September 2025. 2025 (cit. on p. 48).

[105] Laurens van der Maaten and Geoffrey Hinton. «Visualizing Data using t-SNE». In: *Journal of Machine Learning Research* 9.Nov (2008), pp. 2579–2605 (cit. on p. 58).

[106] Daniel Justus, John Brennan, Stephen Bonner, and Andrew Stephen McGough. «Predicting the Computational Cost of Deep Learning Models». In: *2018 IEEE International Conference on Big Data (Big Data)*. 2018, pp. 3873–3882. DOI: 10.1109/BigData.2018.8622396 (cit. on p. 61).