POLITECNICO DI TORINO

Master's Degree course in Electronic Engineering

Master's Degree Thesis

Mixed-Precision Quantization and Inference of MLPerf Tiny DNNs on Precision-Scalable Hardware Accelerators

Politecnico di Torino

Advisors:
prof. Mario Roberto Casu
dott. Luca Urbinati

Candidate:
Marco Alessio Terlizzi

Academic Year 2022-2023

Abstract

Over the past ten years, Deep Learning has made great strides with significant advancements in a variety of Artificial Intelligence (AI) applications that range from image classification to speech recognition. Nevertheless, the unprecedented performance attained by Deep Neural Networks (DNNs) comes at the cost of high computational complexity and power consumption, making them unsuitable for deployment on resource-constrained devices such as embedded hardware. As a result, a field known as TinyML has emerged, aiming to develop efficient and accurate models for the ever-growing market of Internet-of-Things (IoT) devices. Moving both training and inference to the edge offers several advantages, including enhanced data privacy, lower latency, and improved energy efficiency. This is achieved by tackling these issues from multiple angles, such as designing networks that execute fewer operations and reducing the precision of network parameters through quantization. To this regard, this thesis analyzes how mixed-precision quantization can help improve the computational footprint and latency of deep neural networks running on hardware accelerators. First, QKeras, an open-source quantization library, is used to quantize and determine an optimal mixed-precision configuration for four neural network architectures from the MLPerfTiny Benchmark, namely MobilenetV1, Resnet, FC-AutoEncoder, and DS-CNN. Our findings show that this technique is on average able to reduce the number of bits by 59.64% with respect to conventional 16-bit flat quantization techniques while keeping the test accuracy within a range of 2% of their floating-point counterparts. Second, the networks are executed in software on precision-scalable hardware accelerators for DNN algorithms such as 2DConv, DWConv, and FC. In particular, they consist of reconfigurable Sum-Together (ST) multipliers placed inside the MAC units, which make them able to compute N=1,2,4 multiplications in parallel with 16/N bit operands, thus reducing latency when using low precision inputs and weights. These accelerators are designed in C to take advantage of high-level synthesis (HLS) tools. Finally, we also investigate the effects of reducing the bit-width of the accelerator's internal variables, such as the quantization scaling factors, to reduce the accelerators' hardware resources (e.g. multipliers' bitwidths) without degrading the accuracies of the four networks by more than 0.5%. In the process we measure the ideal speedup obtained using the reconfigurable ST multipliers instead of standard ones, finding an average speedup of 2.23x.

Acknowledgements

I would like to express my deepest gratitude to the people who contributed to this thesis.

First and foremost I would like to express my sincere thanks to Prof. Casu, for giving me the opportunity to work on this thesis.

A special thanks goes to Dott. Urbinati, for his patience, motivation and for providing instrumental insight at every stage of this research project.

I extend my heartfelt thanks to my family. Thank you for always supporting me, your constant encouragement has been a driving force behind my academic achievements.

Finally, I am deeply grateful to all the friends who have been by my side throughout this academic journey and made this achievement possible.

Contents

Chapter 1

Background

1.1 Neural Networks and Deep Learning

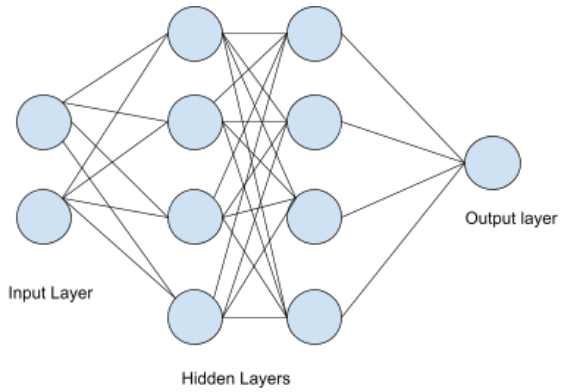


Figure 1.1: Artificial Neural Network [19]

An Artificial Neural Network (ANN) is a system of densely interconnected processing elements loosely inspired by the neural structure of the brain. Unlike traditional computing techniques, ANNs seek patterns in data without the need to be explicitly programmed and have been proven to be very successful at tackling various tasks, ranging from image classification to voice recognition. An ANN is formed by stacking up multiple layers, in which each artificial neuron is fully interconnected with the neurons in the following one, forming a fully connected or dense network. By increasing the number of dense layers, as to create a Deep Neural Network (DNN), the network is able to learn increasingly complex patterns from the input data. A neural network goes through two phases called training and inference. During the first phase the network learns to recognize a specific pattern in a portion of data called training set, while in the second phase the trained neural network generates output predictions from previously unseen input data.

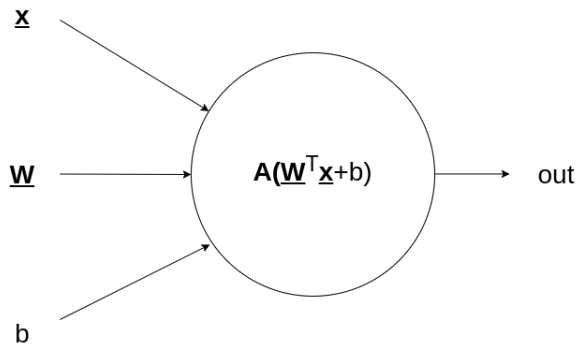


Figure 1.2: Inside a neuron

Taking a closer look at the structure of an artificial neuron (Fig. 1.2), we can see that it is a computational unit that executes a weighted sum, adds a bias, applies a non-linear function and sends the result to neighboring connected neurons. Its parameters are:

- input vector x
- weight vector W
- bias b
- activation function A

Weights and biases are parameters that are "learned" by the network during the training phase and simulate the strengthening of a synaptic pathway. On the other hand, the activation function determines the output of a neuron by introducing a non-linearity, that allows the network to learn complex functions. In classification ANN models a typical activation function for hidden layers is the Rectified Linear Unit, or ReLU (Fig. 1.3), whereas for output layers a common activation is the Softmax function.

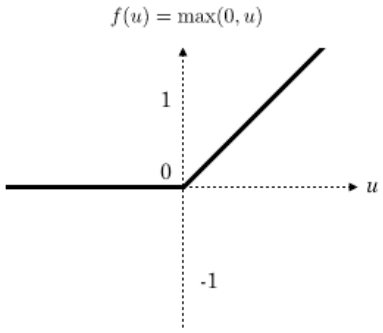


Figure 1.3: ReLU function [34]

Machine learning techniques fall into two basic categories: supervised and unsupervised learning, depending on whether or not the training dataset is labeled, a process that consists in tagging each element in the dataset with the correct prediction value. The network is expected to learn a function that maps the inputs to correct outputs. As the network produces a prediction, this is compared to the expected output. The DNN subsequently uses this information to calculate the loss function, such as "mean squared error" (MSE) as reported in Eq. 1.1, which measures how much the prediction deviates from the ground truth, and to update its weights. The objective of training is to find an optimal configuration of weights and biases that lead to a minimum average loss function, called cost function (J) (Eq. 1.2).

$$L = (y^i - \hat{y}^i)^2 \tag{1.1}$$

$$J = \frac{1}{N} \sum_{i=1}^{N} L(y^i, \hat{y}^i) \tag{1.2}$$

The weight update operation can be achieved by applying an optimization algorithm known as gradient descent. As the name suggests, this involves computing the loss function's gradients with respect to the network's parameters W and b. As shown in Eq. 1.3, by exploiting the graph structure of a neural network, we can use a back-propagation algorithm to calculate the gradients starting from the network's output and recursively applying the chain rule for derivatives up to the input. Gradients are then used to update the parameter's values, after being scaled by the learning rate alpha. Each gradient update is known as a step. A large learning rate helps the network learn faster because of the larger steps, but can lead the cost function to jump to a global minimum or to move away from an optimal minimum. On the other hand, a small value makes the training phase slower and restricts the model to a local minimum, but does not guarantee to find the global minimum because the model may be unable to escape a local one. So it is one of the most important hyperparameters (non-learned parameters) that require fine-tuning to achieve the best training results.

$$\begin{aligned} w &= w - \alpha * \frac{\partial J(w,b)}{\partial w} \\ b &= b - \alpha * \frac{\partial J(w,b)}{\partial b} \end{aligned} \tag{1.3}$$

To determine whether the training phase is performing well, typically, part of the training data is set aside to define a validation set. The algorithm described so far, also known as Batch Gradient Descent, takes one step after the model has done one pass over the entire dataset, which is defined as an epoch. This can be impractical for large datasets and thus the dataset is typically divided into

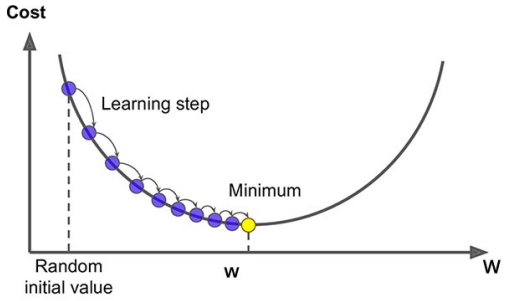


Figure 1.4: Gradient Descent [4]

mini batches, requiring the network to iterate over many of them to complete one epoch. While with Batch Gradient Descent steps are somewhat smooth and converge directly to an optimum solution, with Mini Batch Gradient Descent the overall trend typically fluctuates but converges faster and is less computationally expensive. Once training is complete, the network can be used for inference on a test set to measure how well it does on unseen data.

Although a model might be very accurate on the training set, it is possible that that the network is not able to generalize its knowledge on unseen data. This is a phenomenon commonly known as overfitting (Fig. 1.5) and often comes up in models with many parameters, complex architectures and few training data. In view of this, it is possible to normalize the features of the batch of the input data to set mean 0 and variance 1, in order to stabilize and speed up the learning process. Taking this a step further, it is also possible to normalize the output batch that is generated by hidden neurons with a technique that goes by the name of Batch Normalization (Eq. 1.4). During training, for each batch, we take the mean μ_b and variance σ_b of the features in the batch and we use these two parameters to normalize the features themselves. Instead during inference we use long term mean μ and variance σ values computed as moving averages of batch statistics [24][32]. Batch normalization parameters γ and β are learnable parameters, that are used to scale and shift the normalized activations so that the network is able to use more complex normalization functions that better adapt to the input data.

$$\begin{aligned} x_{bn,train} &= \gamma\left(\frac{x-\mu_b}{\sqrt{\sigma_b^2+\epsilon}}\right)+\beta \\ x_{bn,inference} &= \gamma\left(\frac{x-\mu}{\sqrt{\sigma^2+\epsilon}}\right)+\beta \end{aligned} \tag{1.4}$$

We can exploit the fact that batch normalization parameters μ, σ, γ, β are constants during inference by folding them into the weights and biases of the previous layer (Eqs.1.5) [32]. This is a standard procedure, especially when dealing with inference in embedded devices.

$$W_{folded} = \gamma \frac{W}{\sqrt{\sigma^2 + \epsilon}}$$
$$b_{folded} = \beta - \gamma \frac{\mu}{\sqrt{\sigma^2 + \epsilon}} \tag{1.5}$$

Besides batch normalization, another important technique to limit overfitting is dropout. During training dropout will randomly zero-out the output of selected neurons, forcing the network to learn multiple representations of the input data and thus making it more robust and less likely to overfit.
An additional problem, often encountered during training, is underfitting, a case in which the model is not able to accurately learn the relation between input and output neither during training nor after, thus generating high error rates and poor performance. It is a common issue that arises when the model is too simple, or when there's too little data to train on. The first plot in Fig. 1.5 shows an example of an underfit model, where the fit line is not able to capture the underlying pattern in the data. In contrast, the following plot shows an overfit model whose line fits perfectly the training data but will be unable to generalize to unseen data like the ideal balanced case in the last plot.

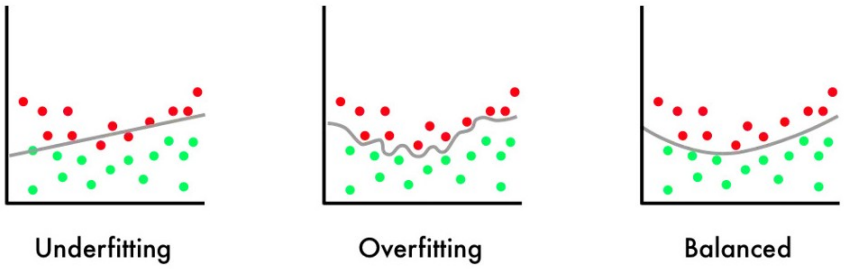


Figure 1.5: Comparison between different fit lines [28]

1.2 Convolutional Neural Networks

1.2.1 Introduction

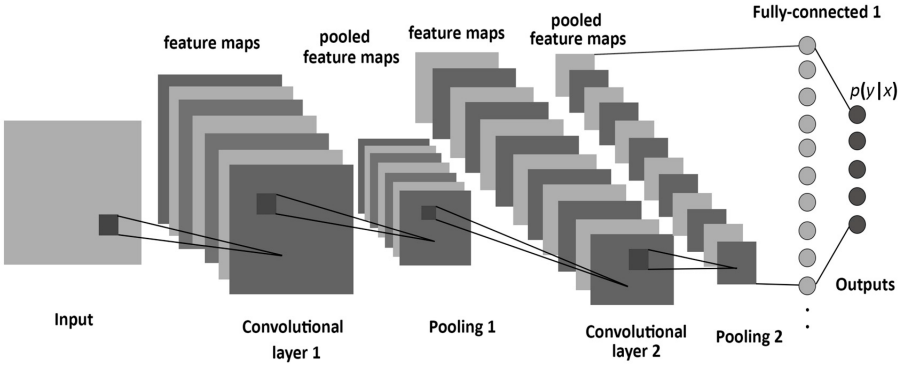


Figure 1.6: CNN architecture example [3]

A common task in machine learning is image classification, i.e. the ability to recognize what is depicted in an image. An image is a multi-dimensional matrix of pixels, also called tensor. When dealing with images having a fully-connected DNN can be rather impractical, if not prohibitive, due to the enormous amount of parameters involved. To illustrate this issue, we can consider an RGB image with a resolution of 1280*720 pixels. This means that a single neuron in the first hidden layer would have 1280*720*3=2764800 weights and, considering that we most likely need more than just one neuron, it easy to see why this approach is not ideal. An additional problem is that we would like our network to recognize objects regardless of their position in the image, a property known as translation invariance. However a fully-connected DNN requires a vector as input, which means that images must be flattened. This effectively destroys the spatial relationship between neighboring pixels and makes it difficult to detect the same feature in different regions of the image. Convolutional Neural Networks (CNNs) solve these issues introducing the concept of locally connected layers and weight sharing. Each output neuron only receives input from a small local portion of the image (Fig. 1.7) called receptive field and learns to extract a feature thanks to a set of weights. We can then reuse these weights to extract the same features in other regions around the image, sliding the weights on the image to create other receptive fields and so other output neurons.

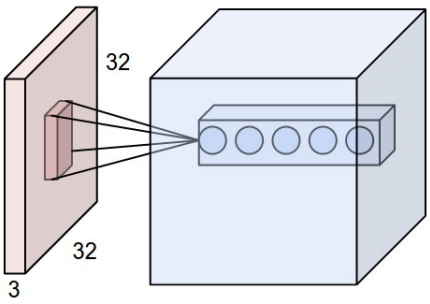


Figure 1.7: Example of a receptive field which generates five output neurons thanks to five filters [43]

The CNN approach not only lets the network learn translation invariant features, but also leads to fewer connections and thus fewer network parameters.

CNNs mainly consist of three types of layers:

- Convolutional layers
- Pooling layers
- Fully-Connected (or Dense) layers

1.2.2 Convolutional layers

In a CNN weights are arranged in 2D structures called kernels, whose width and height determine the receptive field dimensions. A number of kernels equal to the number of input channels is stacked to create a filter. The network uses multiple filters to extract different features from the input tensor. In turn, the number of filters determines the number of channels in the output tensor.
The convolutional layer executes the core operation of CNNs which is a MAC operation: slide a filter across the input, take the element-wise product between the receptive field and each kernel and compute the sum of these products (fig. 1.8). The sum is then fed into an activation function to provide the final output called feature map. For each filter we obtain a different feature map, meaning that for each convolution layer, the input tensor dimensions are modified.
This operation is characterized by the following hyper-parameters:

- filter dimension D_k (e.g. 1x1, 3x3, 5x5, ...)
- number of filters N_f
- stride (S), i-e-the step size
- padding (P), i.e. additional pixels added to the border of each spatial dimension of the input tensor, before convolution is computed

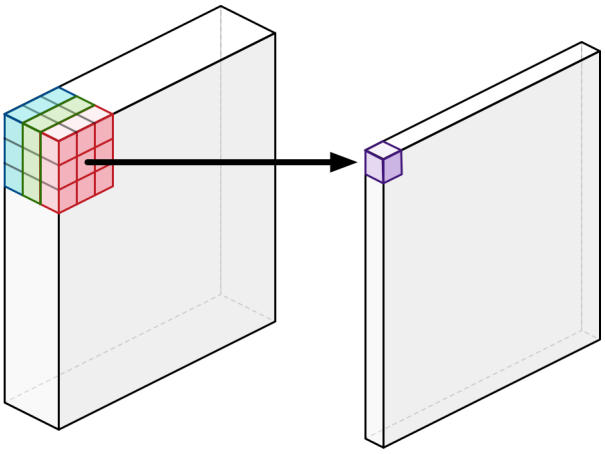


Figure 1.8: 2D convolution between an input tensor with three channels and one filter [15]

0	0	0	0	0	0
0	35	19	25	6	0
0	13	22	16	53	0
0	4	3	7	10	0
0	9	8	1	3	0
0	0	0	0	0	0

Figure 1.9: Zero padding on one channel of a tensor [37]

Applying these concepts to a generic input tensor i with $(H_i \times W_i \times C_i)$ dimensions, the output tensor has $(H_o \times W_o \times C_o)$ dimensions given by the following formulas:

$$\begin{aligned} C_o &= N_f \\ H_o &= \frac{(H_i - D_k + 2P)}{S} + 1 \\ W_o &= \frac{(W_i - D_k + 2P)}{S} + 1 \end{aligned} \tag{1.6}$$

Where H_i and W_i are the input tensor's height and width, H_o and W_o are the output tensor's height and width and C_o is the number of channels in the output tensor.

The total computational cost of this operation in terms of MAC operations [17] is:

$$D_k \times D_k \times C_i \times H_o \times W_o \times C_o \tag{1.7}$$

1.2.3 Pooling Layers

Pooling layers are used to reduce the dimensions of the input tensor. Downsampling the input tensor is important because, not only it reduces the number of parameters to learn but also makes the model more robust to variations in the position of the input tensor's features.

There are two types of pooling layers:

- Max Pooling: outputs the max value of the input region covered by the pooling filter
- Average Pooling: outputs the average value of the input region covered by the pooling filter

Both these layers have stride, padding and dimension as hyperparameters.

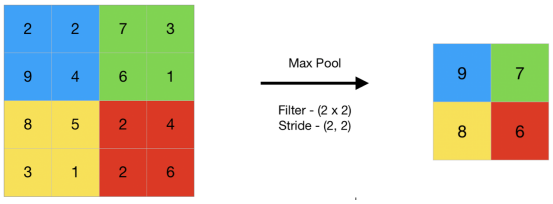


Figure 1.10: Max pool operation [21]

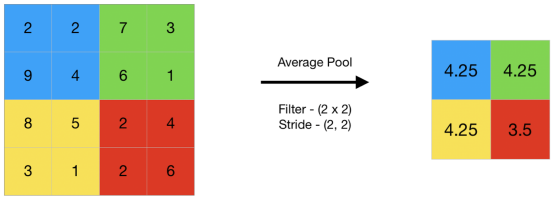


Figure 1.11: Average pool operation [21]

1.2.4 Fully Connected layers

At the end of the network the resulting tensor is finally flattened, creating a vector. A fully connected layer (Fig. 1.12) performs the final classification as described previously in Sec. 1.1.

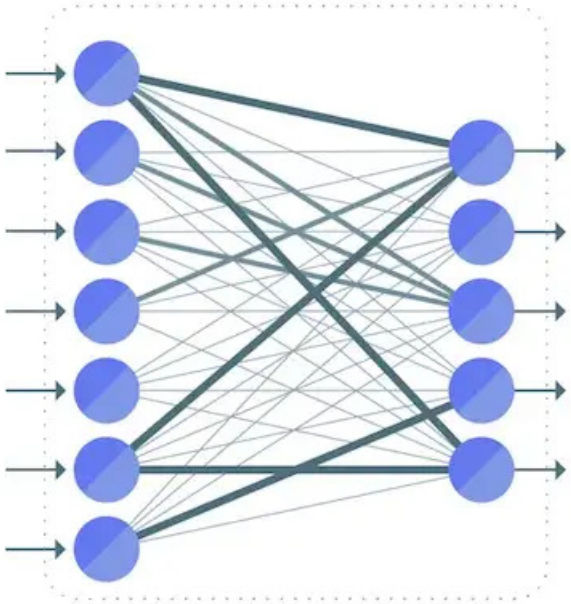


Figure 1.12: Fully-connected layer [18]

This layer implements the following equation:

$$Y = WX + b \tag{1.8}$$

where W $\in \mathbb{R}^{p \times n}$ is the weight matrix, X $\in \mathbb{R}^{m \times p}$ is the input array and b $\in \mathbb{R}^{n}$ the bias vector. The result is Y $\in \mathbb{R}^{m \times n}$, which can also be expressed through Eq. 1.9:

$$Y_{i,j} = b_j + \sum_{k=1}^{p} X_{i,k} W_{k,j} \tag{1.9}$$

1.3 Notable ANN architectures

In this section we give an overview of the network architectures that will be used in this thesis.

1.3.1 Mobilenet

Mobilenet [17] is a CNN architecture that carries out operations in a way that reduces the overall number of parameters while maintaining good accuracy levels. The main idea behind this architecture is the depthwise separable convolution, which is a depthwise convolution followed by a pointwise convolution. Unlike standard convolution, the depthwise convolution does not combine channels, but keeps them separate (Fig. 1.13). This means that the total number of MAC operations is $D_k \times D_k \times C_i \times H_o \times W_o$, using the same notation as in paragraph 1.2.2, because $C_i = C_o$ in the depthwise convolution.

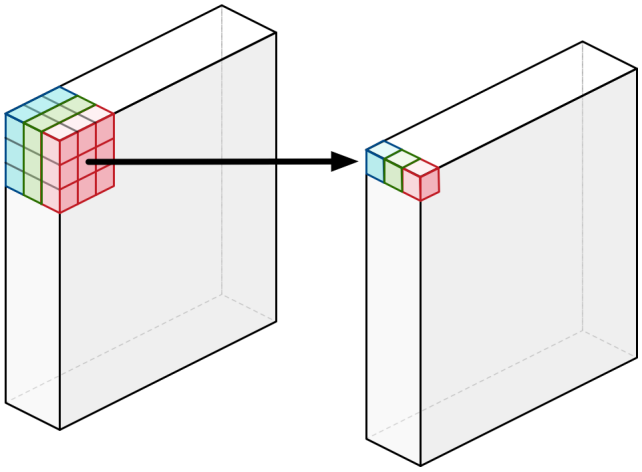


Figure 1.13: Depthwise Convolution [15]

The output channels from the depthwise convolution are then merged through a pointwise convolution, which is a standard convolution with a 1*1 kernel (Fig. 1.14). The number of MACs involved in this operation is $C_i \times C_o \times H_o \times W_o$. The output tensor has the same shape as a standard convolution, but with a drastic reduction in computional cost and just a slight accuracy drop.
The ratio of the depthwise separable convolution MAC cost to the standard convolution is:

$$K = \frac{1}{C_o} + \frac{1}{D_k^2} \tag{1.10}$$

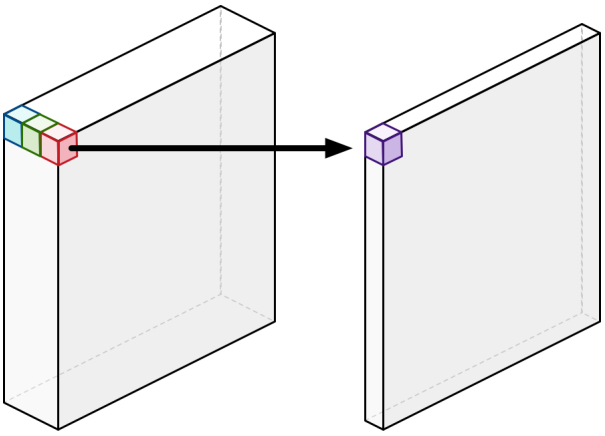


Figure 1.14: Pointhwise Convolution

As a testament to the perfomance of this architecture, if compared to other popular models such as VGG 16, on the ImageNet dataset Mobilenet not only is almost as accurate (71.5% vs 70.6% accuracy,respectively) but also manages to reduce the number of parameters by a factor 32 [17].

1.3.2 Residual Neural Networks

As previously stated, there is a correlation between a network's depth, in terms of layers, and its ability to discern complex patters. Very deep neural networks, however, are rather difficult to train due to the vanishing gradient problem and so suffer from accuracy degradation. Residual Neural Networks (e.g. ResNet) [14] address this problem by introducing skip connections, which in essence are bypass connections (identity mappings) between groups of layers (Fig. 1.15).

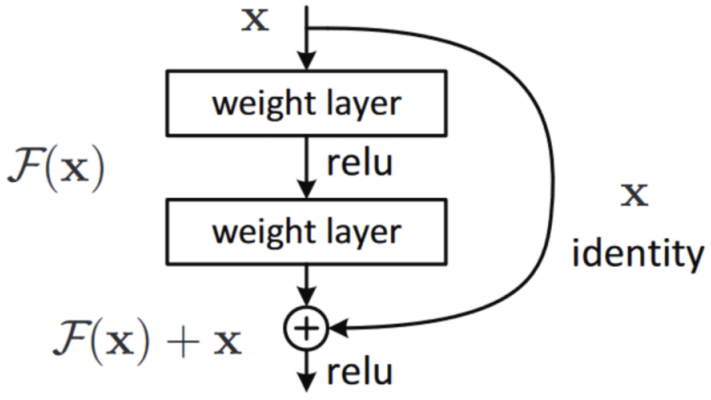


Figure 1.15: An example of a residual/skip connection present in ResNet [14]

The reason why this technique is effective is that the identity function is a way of allowing the network to learn the difference, or residual, between the input and the desired output rather than the output itself. This makes the training process more stable and allows for deeper networks to be trained more effectively, as shown in the original paper [14], which compares the performance of ResNet against a standard convolutional network on the ImageNet dataset when varying the network's depth. It is demonstrated that Resnet improves on the Top-1 error when incrementing the number of layers, while the other network suffers from accuracy degradation.

1.3.3 Auto-encoders

We mentioned unsupervised learning in Sec. 1.1 as a method of training neural networks without the need of labeled datasets, which is often the case in applications such as anomaly detection or fraud detection in which we want to detect rare events and thus lack data. Auto-encoders are a type of neural network that implement this type of learning. An auto-encoder consists of two blocks: an encoder and a decoder (Fig. **??**) . The first block constructs an internal representation of the input, while the second block tries to reconstruct the original representation starting from its internal one (Fig. 1.16). We can then measure the error between the generated representation and the original data, to train the network as usual Sec.1.1. The idea is to try to teach the network to recognize the known and most common events, so that when an outlier event occurs, the network is able to recognize it as anomalous.

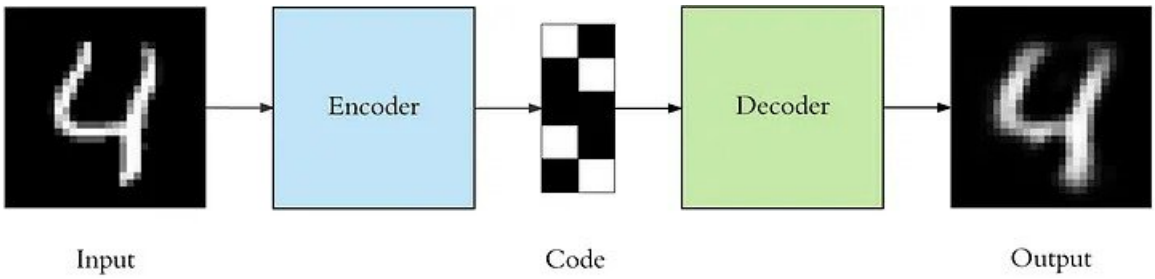


Figure 1.16: Auto-encoder operation [8]

1.4 Neural Network Quantization

Neural network quantization is a technique used to reduce the memory and computation requirements of a network by converting its high-precision floating-point parameters to a lower bitwidth representation. In particular, in this work we will deal with the fixed-point representation. The cost of this operation is the addition of the quantization noise due to rounding and clipping errors, which can lead to a drop in accuracy. There are two approaches to neural network quantization: Post-training quantization (PTQ) and Quantization-aware training (QAT), which will be discussed in Secs. 1.4.2 and 1.4.3.

1.4.1 Affine quantization schemes

To map a floating point number to an integer representation, we can define the following parameters [24][32]:

- bitwidth b
- scale factor s
- zeropoint z

The bitwidth defines the integer grid size (the bits needed to represent a number after quantization), the scale factor defines the quantizer's step-size and the zeropoint makes sure the value zero is quantized with no error, which is paramount for operations such as zero-padding. These parameters can be defined per-tensor or per-channel. Usually the latter is used when the tensor that is being quantized has values that vary greatly between its channels.

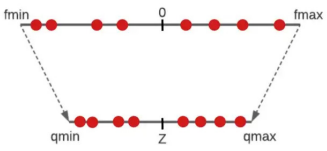


Figure 1.17: Quantization mapping

From a mathematical standpoint, mapping the floating point range $[\alpha, \beta]$ to the quantized range $[\alpha_q, \beta_q]$ requires solving a linear system which leads to the following definitions for scale and zeropoint [31]:

$$\begin{aligned} s = scale &= \frac{\beta - \alpha}{\beta_q - \alpha_q} \\ z = zeropoint &= \frac{\alpha\beta_q - \beta\alpha_q}{\beta - \alpha} \end{aligned} \tag{1.11}$$

The quantization (Eq. 1.12) and de-quantization (Eq. 1.13) operations are defined by the following equations [32]:

$$x_q = clamp(round(x_{float}/s) + z, 0, 2^b - 1) \tag{1.12}$$

$$x_{dq} \approx \mathrm{s}(\mathrm{x}_q - z) \tag{1.13}$$

This quantization scheme is also known as "asymmetric uniform affine", while if we restrict the zeropoint to the value 0, we get the "symmetric uniform affine" quantization. This type of quantization is well-suited for a weight distribution which is roughly symmetrical around zero. On the other hand, if it is skewed towards positive or negative values, the best option would be the asymmetric quantization scheme.
The choice of quantization scheme has an impact on the computational overhead. This can be seen by applying Eq. 1.12 to Eq. 1.9, which leads to Eq. 1.14 [31]:

$$\begin{aligned} Y_{q,i,j} &= z_Y + \frac{s_b}{s_Y}(b_{q,j} - z_b) \\ &+ \frac{s_X s_W}{s_Y}\left[\left(\sum_{k=1}^{p} X_{q,i,k} W_{q,k,j}\right) - \left(z_W \sum_{k=1}^{p} X_{q,i,k}\right) - \left(z_X \sum_{k=1}^{p} W_{q,k,j}\right) + p z_X z_W\right] \end{aligned} \tag{1.14}$$

where z_w, z_b, z_X, z_Y and s_w, s_b, s_X, s_Y are the zero points and scales of the weights, biases, inputs and outputs, respectively. The formula can be simplified by forcing the zero points of the weights and biases to zero, during training, which is the case when both the quantized range and floating-point range of weights and biases are symmetric. This simplification allows for a more hardware friendly implementation of the quantization operation [20]. Determining an accurate quantization range is a fundamental step to ensure that no unnecessary noise is added. For weights (and also for biases) this is a relatively simple task [20]. Since they are known constants at inference time, we can set:

$$\begin{aligned} \alpha &:= min(W) \\ \beta &:= max(W) \end{aligned} \tag{1.15}$$

Activations, on the other hand, depend on the input, which is not known a priori during inference time. So typically a calibration dataset is used to estimate a good set of parameters via exponential moving averages [20] or other techniques such as Mean Squared Error (MSE) [32].

Another important step is choosing an appropriate quantization granularity. Typically weights and biases are quantized per-channel, while activations are quantized per-layer as it is not possible to factor the scale out of the summation which complicates the hardware implementation [24][32].

1.4.2 PTQ

Post-training quantization is the fastest approach to quantize a neural network and consists in applying quantization to the pre-trained FP32 network parameters. This approach is simple and works adequately for large models, where it is possible to achieve near floating-point accuracy using 8 bit quantization schemes and per-channel quantization [24]. The most critical aspect of this method is finding good quantization parameters. This technique however falls short of expectations when applying lower bit quantization schemes, and when applying per-layer quantization. For this reason QAT proves to be a more effective solution.

1.4.3 QAT

Post-training techniques may not be enough to mitigate the large quantization error incurred by low-bit quantization and low-granularity quantization. QAT is a method that usually allows the quantized network to reach FP accuracy also in these conditions [24]. This method adds a simulated quantization noise during the training phase, so that the network learns to counteract it. Starting with a pre-trained floating point network, the effect of quantization noise can be modeled in the forward pass by applying simulated quantization operations on the network's weights, biases and activations. The backward pass is also modified because the quantized layers are not differentiable. To overcome this issue, a straight through estimator [32] is used to pass on the incoming gradient as if the function were an identity function inside the clipping range [31] (1.18).

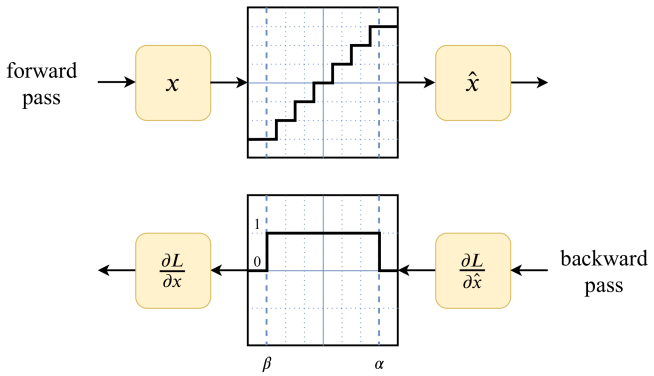


Figure 1.18: Straight through estimator [31]

Chapter 2

Neural Network Quantization with AutoQKeras

In this chapter we introduce QKeras [7], a quantization library designed as an extension of the Keras API [1], and the AutoQKeras Python class, that enables a smart selection of the layers' bitwidths using Bayesian Optimization [10]. Furthermore, in order to learn the different features of these tools and establish a starting point for the work presented in the following chapters, we conduct a preliminary study by quantizing a simple neural network in mixed-precision using [**?**].

2.1 QKeras

QKeras is an open source quantization library developed by Google that introduces replacement layers for Keras allowing for a straightforward and fast deployment of quantized models. QKeras layers inherit characteristics from their floating-point Keras counterparts and expand upon them introducing new functionalities that enable QAT. Generally, layers can be categorized into two classes, depending on whether they do data manipulation or just data transport. The former category refers to layers, such as QConv2D, QDepthwiseConv2D and QDense, that apply some arithmetic operation, and thus require the user to specify a quantization policy (called *quantizer* in QKeras) for weight and bias. Likewise, activation layers require a quantizer, and are implemented as a QActivation layer that merges quantization and activation functions. In contrast, data transport layers like Flatten, Input or Output do not require quantizers as they just modify the shape of the input tensor and pass it down the data stream. Another layer that does not require quantizers is MaxPooling since the output feature map values will always fall into

the quantization grid determined by the bit precision of the network.
QKeras also implements Batch Norm folding by automatically scanning for QConv2D/QDepthwiseConv2D, Batch Normalization pairs and folding them into one layer called QConv2DBatchnorm/QDepthwiseConv2DBatchnorm. By default these will be implemented following the Exponential Moving Average (EMA) discussed in [20]. The function first computes folded weights and biases, then quantizes them and finally computes a standard Keras convolution (Conv2D/DepthwiseConv2D).

QKeras provides several quantizer options, that range from binary quantization functions for weights to quantized implementations of relu and sigmoid activation functions [6]. A quantizer that is of particular interest is *quantized_bits*, a class that executes a uniform affine quantization mapping equal to that of Eqs. 1.12-1.13. It takes the following arguments:

- *bits*: total number of bits on which to map the FP representation
- *integer*: the number of bits in the integer part, according to a fixed-point representation. Must be lower or equal to *bits*
- *symmetric*: sets symmetric clipping ranges (i.e. $alpha_q = -beta_q$
- *keep_negative*: if true the quantized range spans both positive and negative ranges
- *alpha*: mode of operation ("1", "auto" or "auto-po2", explained later)
- *scale_axis*: which axis of the FP tensor to calculate the scale from. Scale_axis = 0 applies per-layer quantization

When the mode of operation, *alpha*, is an integer (typically "1"), the function computes the following:

$$\begin{aligned} xq &= k \times \frac{m_i}{m} \\ &\times clip(round(x \times \frac{m}{m_i}), keep\_negative \times (-m + symmetric), m - 1) \end{aligned} \tag{2.1}$$

where $m_i = 2^{integer}$, $m = 2^{(bits - keep\_negative)}$, x is the floating point value, and k [1] is equal to the integer value of the class attribute *alpha*. This mode of operation can be used to train the network using a fixed-point quantization. However there is no scaling factor that compensates variations of *integer*. On the other hand, by

[1] It should be noted that QKeras refers to the parameter k as "alpha", same as the class attribute *alpha*. To avoid ambiguity we refer to this parameter as k.

setting alpha to *auto*, *quantized_bits* will compute the uniform affine quantization mapping as Eq. 1.12. No matter what *integer* value is passed as argument to the class, the scale (Eq. 2.2) will adjust accordingly, ensuring that the output will be the same, regardless of *integer*.

$$k = scale = \frac{2max(|x|)}{\beta_q - \alpha_q} \tag{2.2}$$

Finally when *alpha* is "auto_po2" QKeras approximates the scale from Eq. 2.2 to the nearest power of 2 number, resulting in some accuracy loss but a simpler hardware implementation.

It is important to note that when *alpha* is *auto*, *quantized_bits* only implements symmetric uniform affine quantization, i.e. a symmetric quantized range, by automatically forcing *symmetric* and *keep_negative* to 1 internally, which in turn results in the loss of one bit in the positive range. Moreover, by implementing the scale as in Eq. 2.2 instead of the more general Eq. 1.11, QKeras eliminates the contributions z_w and z_b from Eq. 1.14.

In conclusion, QKeras implements QAT applying the "fake quantization" technique shown in Sec. 1.4.3 by means of quantize-dequantize operations and STE for back-propagation. Layers such as QDense, QConv2D and QDepthwiseConv2D contain floating-point weights from Keras, whose weights are constrained to use quantized values obtained with a quantization-dequantization operation. However, it is important to note that their input feature map tensors are not quantized and so remain in FP format. One solution already included in QKeras is to replace the standard ReLU layers from Keras with QActivations layers from QKeras with *quantized_relu* as quantizer, in order to quantize-dequantize the feature maps between two layers. However, this way QKeras does not use the affine quantization method described previously. A possible workaround is to use a custom quantizer for affine quantization and pass it to the QActivation layer. Thus in this thesis we developed *quantized_bits_featuremap*, a class inspired by *quantized_bits*, which implements the zeropoint which was missing in the original QKeras implementation. Now QKeras completely supports affine quantization, both for weights (using the original quantized_bits class) and activations (using the quantized_bits_featuremap class instead of quantized_relu)

2.2 Keras Tuner and AutoQKeras

Finding the best tradeoff between bit reduction and network performance is a hyperparameter search problem that QKeras addresses with AutoQKeras. This Python class enables the automatic exploration of the search space by treating this problem as a Keras Tuner [33] hyperparameter optimization problem. Keras Tuner is another Python library that requires the user to define a hyperparameter search space and will try to find a hyperparameter configuration, known as a hypermodel, that maximizes a certain metric, such as validation accuracy. The elements in the search space are sampled according to a search algorithm which, in AutoQKeras, can be either Random Search, Hyperband [27] or Bayesian Optimization [10]. All these algorithms perform iterations, known as trials, in which elements from the hyperparameter search space are sampled and used to train a model. A metric called *score* is associated to each trial to measure its performance relative to other trials.

Random search, as the name suggests, will randomly sample a set of hyperparameters, typically choosing from a uniform distribution. Due to the random nature of this algorithm, there is no way of knowing if an optimal solution was found. Nevertheless it offers a simple way to explore the hypermodel space as it is easy to implement and makes no assumptions about the structure of the space.

Hyperband, expands upon random search by integrating ideas from the successive halving algorithm. The core idea of the successive halving algorithm is to start training n hypermodels with a fixed resource budget B (such as the number of epochs), and progressively discard the worst performing ones. One of the challenges is to find the best tradeoff between B and n, or, in other words, determining for a fixed budget if it's better to consider many models (large n) with a small training time or a few models with a longer training time. Hyperband addresses this issue allocating a fixed budget B for several values of n in what is called a bracket, which is essentially a successive halving round. For each round the algorithm will discard the worst performing ones and increasingly allocate more budget to the next one, until it finds the best solution. In spite of the fact that it was shown to perform better than Random Search [27], it still comes with the shortcomings related to the initial random hyperparameter sampling.

Finally, Bayesian Optimization is performed by building a probabilistic model of the objective function, typically the model's accuracy, called surrogate which is then used to guide the search for the optimal set of hyperparameters. While computationally intensive, given enough trials this technique will eventually converge to an optimum hypermodel, since it iteratively evaluates new hyperparameters based on past performance.

2.3 AutoQKeras

Given a DNN with a fixed architecture, in AutoQKeras the hyper-parameter search space is defined by the quantizers that are chosen for each layer and their number of bits. Since quantization degrades accuracy, it is not possible to directly optimize the search for this metric, as the algorithm would most likely choose from quantizers having many bits. The objective, however, is to find a tradeoff between performance and resource utilization. To avoid this, AutoQKeras defines a metric called *forgiving factor* (FF) [7], defined as:

$$FF = 1 + \Delta * log_{rate}(stress * \frac{reference\_cost}{trial\_cost}) \tag{2.3}$$

where *rate* and *stress* are arbitrary constants, while Δ is a function of the reference and trial cost, which can be either equal to *delta_p* or *delta_n* (two user defined constant parameters). In particular $\Delta = delta\_p$ if the trial cost is lower than the reference cost (defined later), while $\Delta = delta\_n$ in the opposite case. The network's cost is determined by the optimization goal, which can be either "energy" or "bit". So it is a measure of the number of bits or energy consumption of the network. In particular, the energy cost is an estimate based on [16]. However, it's not a quantitative measure, as energy cost depends on many factors including technology. Due to this specific reason, from now on we will only consider bit optimization, because it is a more technology independent metric. AutoQKeras defines two costs: the trial cost refers to the cost of the particular model chosen during a specific iteration, in other words, the total number of bits of the quantized model. The reference cost is the cost of a model with a fixed number of bits for each layer, that is used by AutoQKeras as a reference to compare to different models.

The tool is configured through a goal dictionary where the user can set the following parameters:

- *type*: "bits" (or energy)
- *delta_p*: constant that sets Δ
- *delta_n*: same as above
- *rate*: constant used for tuning
- *stress*: same as above
- *input_bits*: constant that sets the number of bits for the Input layer, used to compute the trial and reference cost
- *output_bits*: constant that sets the number of bits for the output layer (i.e. Softmax), used to compute the trial and reference cost

- *ref_bits*: constant that sets the number of bits to assign to all layers (except input and output) in the reference model
- *config*: specifies whether to take into account both the activation and parameter bits or just one of the two

Internally, the bit cost is computed as:

$$tot\_bits = num\_parameters \times p\_bits + num\_activations \times a\_bits \tag{2.4}$$

where *p_bits* and *a_bits* are the number of bits chosen for the parameters and activations, respectively.
To sum up, the FF (Eq. 2.3) is used to boost the model's accuracy so that Keras Tuner is able to select a smaller network. AutoQKeras achieves this by taking the the ratio of the reference cost to the trial cost and uses this value, save for multiplicative and additive constants, as a boosting factor for the accuracy, resulting in the final score function which is the metric maximized by the AutoQKeras research.

$$score = accuracy * FF \tag{2.5}$$

In order to counteract accuracy degradation in quantized models, AutoQKeras also provides optional filter-tuning capabilities that will increment the number of filters when deeply quantizing to low bit-widths. Furthermore the user can specify which specific layers to quantize and also the maximum number of bits to use each layer. In this work filter tuning will always be disabled.

2.4 Preliminary experiments with AutoQKeras

In this section we explore and apply AutQKeras' functionalities to a simple neural network (Tab. 2.1) that can be found in the AutoQKeras notebook AutoQKeras.ipynb [12] The goal is to determine a good set of parameters to use in subsequent experiments. The notebook was executed on the free version of Google Colab.

Model: model

Layer (type)	Output Shape	Param #
input (InputLayer)	[(None, 28, 28, 1)]	0
conv2d_0 (Conv2D)	(None, 14, 14, 16)	144
bn_0 (BatchNormalization)	(None, 14, 14, 16)	64
act_0 (Activation)	(None, 14, 14, 16)	0
drop_0 (Dropout)	(None, 14, 14, 16)	0
conv2d_1 (Conv2D)	(None, 7, 7, 32)	4608
bn_1 (BatchNormalization)	(None, 7, 7, 32)	128
act_1 (Activation)	(None, 7, 7, 32)	0
drop_1 (Dropout)	(None, 7, 7, 32)	0
conv2d_2 (Conv2D)	(None, 4, 4, 48)	13824
bn_2 (BatchNormalization)	(None, 4, 4, 48)	192
act_2 (Activation)	(None, 4, 4, 48)	0
drop_2 (Dropout)	(None, 4, 4, 48)	0
conv2d_3 (Conv2D)	(None, 2, 2, 64)	27648
bn_3 (BatchNormalization)	(None, 2, 2, 64)	256
act_3 (Activation)	(None, 2, 2, 64)	0
drop_3 (Dropout)	(None, 2, 2, 64)	0
conv2d_4 (Conv2D)	(None, 1, 1, 128)	73728
bn_4 (BatchNormalization)	(None, 1, 1, 128)	512
act_4 (Activation)	(None, 1, 1, 128)	0
drop_4 (Dropout)	(None, 1, 1, 128)	0
flatten (Flatten)	(None, 128)	0
dense (Dense)	(None, 10)	1290
softmax (Activation)	(None,10)	0
Total params: 122,394		
Trainable params: 121,818		
Non-trainable params: 576		

Table 2.1: Reference floating point model **??**

The model is trained on the MNIST dataset [9], a collection of 28x28 greyscale images of handwritten digits split into 60000 train samples and 10000 validation samples. The reference floating-point model is able to reach 99.37% accuracy on the validation set, after training for 200 epochs with a batch size of 4096 samples and using the Adam Optimizer with constant learning rate equal to 0.02 . The quality of the results of three different AutoQKeras configurations were tested, one for each of the Keras Tuner search optimizations available: Random, Bayesian and Hyperband.
The weights and biases search space was defined in this way, meaning that each weight/bias tensor of each layer of the network could be quantized with one of these quantizers at each search iteration (trial) The search space for weight and bias was defined as: quantized_bits(X,Y,1,alpha=1), where X is 16, 8 or 4, while Y is in the range 0 to 15 The activation search space was defined in a similar fashion, using *quantized_relu*, a quantized implementation of the ReLU function, instead of *quantized_bits*
Each search optimization tuner was executed three times for 20 trials, each time limiting the maximum number of bits to 16, 8 and 4, resulting in the three search spaces called *LIMIT16*, *LIMIT8* and *LIMIT4*. The best hypermodels were then re-trained from scratch for 200 epochs and a batch size of 4096.

The AutoQKeras configuration dictionary is reported in Tab. 2.2:

The *goal_bits* parameter is a Python dictionary that specifies *bit* as the optimization goal and sets the FF parameters:

- delta_p: 5.0
- delta_n: 5.0
- rate: 2.0
- stress: 1.0
- input_bits: 8
- output_bits: 8
- ref_bits: 16
- config: ["parameters","activations"]

Table 2.3 summarizes the results and shows the bit configuration of each layer in the form $< W.I >$, where W is the total number of bits and I is the integer part. Several observations can be made by analyzing these results: considering that the model at issue is very simple, all tuners are able to achieve good accuracies. Nevertheless, there are significant differences in the total time required for searching the quantization space and the results obtained by the tuners.

	Random	**Bayesian**	**Hyperband**
output_directory	random	bayesian	hyperband
goal	goal_bits	goal_bits	goal_bits
learning_rate_optimizer	False	False	False
transfer_weights	False	False	False
mode	random	bayesian	hyperband
seed	42	42	42
limit	16 8 4	16 8 4	16 8 4
tune_filters	None	None	None
distribution_strategy	cur_strategy	cur_strategy	cur_strategy
layer_indexes	range (1,len(model.layers)-1)	range (1,len(model.layers)-1)	range (1,len(model.layers)-1)
max_trials	20	20	20
factor	-	-	3
hyperband_iterations	-	-	1

Table 2.2: AutoQKeras configuration specs for the experiment on the network in Tab. 2.1

As expected Bayesian Optimization, being the most computationally intensive of the three, is the slowest, followed by Hyperband and then Random Search. The best hypermodels in terms of bits were found by the Bayesian tuner, which in both 16 and 8 bit limit cases tends to choose 4 bits for each layer, without degrading accuracy by more than 1%. Conversely, Random Search and Hyperband produce comparable outcomes, not performing as well as Bayesian Optimization in the 16 bit limit case, though noticeably faster. It should be noted that the 4 bit limit category is a particular case in which all quantizers are set to 4 bit, for this reason the total bits are the same for all tuners. Nevertheless, the integer part of the quantizer varies, and it shows how Bayesian Optimization is able to tune the integer part to obtain better results in terms of validation accuracy with respect to the other two tuners.

All in all, this preliminary study shows that the Bayesian tuner, as expected, outperforms Random and Hyperband tuners in terms of result quality. For these reasons Bayesian Optimization will be adopted as the standard tuner for all subsequent AutoQKeras experiments in this thesis

	LIMIT 16			LIMIT 8			LIMIT 4		
	random	bayesian	hyperband	random	bayesian	hyperband	random	bayesian	hyperband
training accuracy [%]	99.37	99.05	98.78	99.09	99.08	97.98	99.14	99.05	96.76
validation accuracy [%]	99.39	99.15	99.08	99.27	99.08	98.87	99.15	99.36	98.6
val acc degradation [%]	+0.02	-0.22	-0.29	-0.10	-0.29	-0.50	-0.22	-0.01	-0.77
search time [s]	731.67	928.40	669.55	738.73	936.38	647.88	741.93	958.01	641.71
total bits (weights only)	833832	484968	840336	503440	485584	601256	484968	484968	484968
BEST MIXED PRECISION CONFIGURATION									
conv2d_0	16.3	4.0	4.1	4.0	8.8	8.6	4.2	4.1	4.1
bn_0									
act_0	8.6	4.0	16.9	8.6	4.4	8.5	4.4	4.3	4.4
conv2d_1	16.1	4.0	16.2	8.3	4.0	4.2	4.2	4.2	4.1
bn_1									
act_1	16.4	16.16	8.2	4.4	4.0	4.2	4.4	4.4	4.3
conv2d_2	16.1	4.0	4.2	4.1	4	4.3	4.2	4.4	4.3
bn_2									
act_2	16.15	4.0	4.0	8.7	4.0	8.6	4.2	4.4	4.1
conv2d_3	8.1	4.0	4.0	4.0	4.0	8.2	4.4	4.4	4.3
bn_3									
act_3	8.8	4.0	16.1	8.7	8.4	4.2	4.4	4.4	4.4
conv2d_4	4.4	4.0	8.0	4.2	4.0	4.2	4.2	4.4	4.3
bn_4									
act_4	8.4	4.0	8.5	8.0	4.0	8.0	4.3	4.0	4.4
dense [a]	k =16.10 b = 4.0	k = 4.0 b = 4.0	k = 8.2 b = 8.5	k = 4.3 b = 8.0	k = 4.0 b = 8.8	k = 8.1 b = 4.1	k = 4.0 b = 4.4	k = 4.0 b = 4.4	k = 4.0 b = 4.0

Table 2.3: Results obtained from the preliminary experiment

[a]This model's Dense layer has both kernel k and bias b. The other layers do not have bias vectors

2.5 Extending AutoQKeras features

Despite being implemented in QKeras, batch normalization folding was not integrated in AutoQKeras' source code. In particular, the AutoQKeras class *init* function was missing the *enable_bn_folding* parameter that enables batch normalization folding when calling the *model_quantize* function. This means that in AutoQKeras this parameter is fixed to its default value, which is *False*. A first trivial step consists in making this parameter visible in the AutoQKeras class interface. While this effectively enables folding, it does not consider that folded layers, namely QConv2dBatchnorm and QDepthwiseConv2dBatchnorm, are ignored by the *forgiving_bits* class, which is responsible for computing the bit size of the reference and trial models [11]. Thus, to avoid underestimating the total size of the trial model when *enable_bn_folding* = *True*, the class was updated so that it could correctly compute the size of these layers.
Another matter to consider was the *transfer_weights* parameter in the *model_quantize* function, which facilitates the transfer of pre-trained weights from a Keras model to the quantized QKeras model. In the original source code, *transfer_weights* and *enable_bn_folding* cannot be enabled at the same time. The reason for this is that Keras layers with weights (e.g. Conv2D) have only two parameters inside the weight tensor, weights and biases. On the other hand folded layers also include the four batch normalization parameters μ, σ, γ, β, plus a parameter called *iteration*. AutoQKeras will attempt to convert the FP Keras model to a quantized QKeras model with folded layers, but will fail to transfer weights as these layers expect seven parameters instead of just two. To address this issue, during weight transfer the folded layer receives weights and biases from the Keras model, while maintaining its own batch normalization parameters.

2.6 Mobilenet quantization

An additional experiment was conducted using Bayesian Optimization on the version of MobileNetV1 provided by MLPerfTiny [36] using the Visual Wake Words dataset [5]. This network and dataset will be discussed in detail in the following chapter. Pre-trained weights from the MLPerfTiny Benchmark [36] repository on GitHub were used as a starting point for training, meaning that only a few epochs of fine tuning were necessary to recover the accuracy, which is 86% on the test set for the floating-point Keras model. The test set was selected using 1000 image indexes provided by MLPerfTiny in the evaluation directory of the official GitHub repository [2]. [2]

The AutoQKeras search space is the same as the previous experiment, but it was reduced by skipping odd bit numbers, an arbitrary choice justified by the need to halve the enormous amounts of combinations to be tried by the tuner. The experiment was executed on Politecnico di Torino's Nvidia GTX 1070 GPU, and lasted approximately 48 hours, and executed a total of 219 trials. Each trial consisted in training a specific hypermodel for 10 epochs, discarding the worst-performing ones as soon as possible using early stopping with patience = 3 and min_delta = 0.01 (i.e training is stopped whenever validation accuracy does not improve by 0.01 in three consecutive epochs). AutoQKeras' results, summarized in Fig. 2.3, show that the best model of the BO search (trial 167) reduces the memory footprint by 33.71% with respect to a flat 16-bit quantized model (4866150 vs 7340702) bits [3]). The quantization configuration is shown in Fig. 2.1.

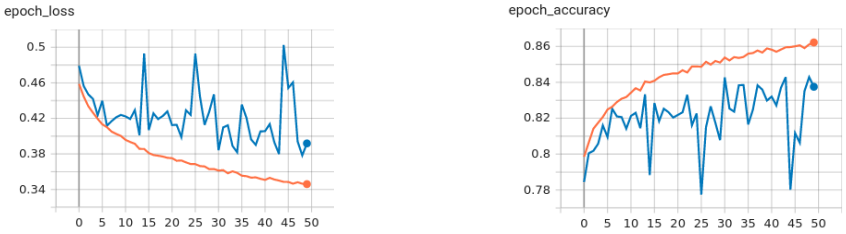
The model was then re-trained, starting from the weights of the best epoch of the best trial, using the same training configurations as the script provided by MLPerf Tiny (see Sec. 3.5 for an in-depth explanation). This model, evaluated on the 1000 test images, achieves 85.79 % accuracy (best model from epoch 49 in 2.2), that is a 0.21% loss in accuracy with respect to the FP pre-trained model.

[2]The actual test set that MLPerf uses to evaluate the model is not publicly available [40]. More on this in Sec. 3.5.

[3]AutoQKeras computes the total number of bits using Eq. 2.4 while the validation score comes from Eq. 2.5

```
stats: delta_p=0.05 delta_n=0.05 rate=2.0 trial_size=4866150 reference_size=7340702
       delta=2.97%
       a_bits=2598928/3926032 (-33.80%) p_bits=2267222/3414670 (-33.60%)
       total=4866150/7340702 (-33.71%)
conv2d                f=8 quantized_bits(16,1,1,alpha=1.0) quantized_bits(31,29,1,alpha=1.0)
activation            quantized_relu(16,3)
depthwise_conv2d      f=None quantized_bits(8,1,1,alpha=1.0) quantized_bits(16,0,1,alpha=1.0)
activation_1          quantized_relu(4,0)
conv2d_1              f=16 quantized_bits(16,7,1,alpha=1.0) quantized_bits(16,13,1,alpha=1.0)
activation_2          quantized_relu(8,2)
depthwise_conv2d_1    f=None quantized_bits(16,13,1,alpha=1.0) quantized_bits(16,7,1,alpha=1.0)
activation_3          quantized_relu(16,5)
conv2d_2              f=32 quantized_bits(16,11,1,alpha=1.0) quantized_bits(31,13,1,alpha=1.0)
activation_4          quantized_relu(16,3)
depthwise_conv2d_2    f=None quantized_bits(8,1,1,alpha=1.0) quantized_bits(31,1,1,alpha=1.0)
activation_5          quantized_relu(4,3)
conv2d_3              f=32 quantized_bits(16,13,1,alpha=1.0) quantized_bits(16,15,1,alpha=1.0)
activation_6          quantized_relu(8,8)
depthwise_conv2d_3    f=None quantized_bits(8,1,1,alpha=1.0) quantized_bits(31,27,1,alpha=1.0)
activation_7          quantized_relu(16,15)
conv2d_4              f=64 quantized_bits(4,1,1,alpha=1.0) quantized_bits(31,7,1,alpha=1.0)
activation_8          quantized_relu(8,6)
depthwise_conv2d_4    f=None quantized_bits(8,3,1,alpha=1.0) quantized_bits(16,15,1,alpha=1.0)
activation_9          quantized_relu(8,6)
conv2d_5              f=64 quantized_bits(16,11,1,alpha=1.0) quantized_bits(16,3,1,alpha=1.0)
activation_10         quantized_relu(16,15)
depthwise_conv2d_5    f=None quantized_bits(16,5,1,alpha=1.0) quantized_bits(16,7,1,alpha=1.0)
activation_11         quantized_relu(8,4)
conv2d_6              f=128 quantized_bits(4,1,1,alpha=1.0) quantized_bits(31,7,1,alpha=1.0)
activation_12         quantized_relu(16,13)
depthwise_conv2d_6    f=None quantized_bits(8,1,1,alpha=1.0) quantized_bits(16,13,1,alpha=1.0)
activation_13         quantized_relu(16,13)
conv2d_7              f=128 quantized_bits(16,7,1,alpha=1.0) quantized_bits(16,0,1,alpha=1.0)
activation_14         quantized_relu(8,6)
depthwise_conv2d_7    f=None quantized_bits(16,3,1,alpha=1.0) quantized_bits(16,11,1,alpha=1.0)
activation_15         quantized_relu(4,3)
conv2d_8              f=128 quantized_bits(16,3,1,alpha=1.0) quantized_bits(31,3,1,alpha=1.0)
activation_16         quantized_relu(8,6)
depthwise_conv2d_8    f=None quantized_bits(8,5,1,alpha=1.0) quantized_bits(31,7,1,alpha=1.0)
activation_17         quantized_relu(16,11)
conv2d_9              f=128 quantized_bits(16,5,1,alpha=1.0) quantized_bits(31,11,1,alpha=1.0)
activation_18         quantized_relu(16,3)
depthwise_conv2d_9    f=None quantized_bits(16,5,1,alpha=1.0) quantized_bits(31,13,1,alpha=1.0)
activation_19         quantized_relu(16,3)
conv2d_10             f=128 quantized_bits(8,5,1,alpha=1.0) quantized_bits(31,27,1,alpha=1.0)
activation_20         quantized_relu(8,0)
depthwise_conv2d_10   f=None quantized_bits(4,1,1,alpha=1.0) quantized_bits(31,25,1,alpha=1.0)
activation_21         quantized_relu(8,0)
conv2d_11             f=128 quantized_bits(16,5,1,alpha=1.0) quantized_bits(16,5,1,alpha=1.0)
activation_22         quantized_relu(8,2)
depthwise_conv2d_11   f=None quantized_bits(4,0,1,alpha=1.0) quantized_bits(16,11,1,alpha=1.0)
activation_23         quantized_relu(8,4)
conv2d_12             f=256 quantized_bits(16,9,1,alpha=1.0) quantized_bits(16,5,1,alpha=1.0)
activation_24         quantized_relu(4,1)
depthwise_conv2d_12   f=None quantized_bits(4,1,1,alpha=1.0) quantized_bits(31,5,1,alpha=1.0)
activation_25         quantized_relu(16,9)
conv2d_13             f=256 quantized_bits(4,0,1,alpha=1.0) quantized_bits(31,15,1,alpha=1.0)
activation_26         quantized_relu(16,15)
dense                 u=2 quantized_bits(4,1,1,alpha=1.0) quantized_bits(31,27,1,alpha=1.0)
```

Figure 2.1: Best mixed-precision MobileNet configuration found by AutoQKeras with alpha=1 (Trial 167 in Fig. 2.3)



(a) Loss **(b)** Accuracy

Figure 2.2: Training results of the best model (trial 167) of MobileNet V1 obtained from AutoQKeras with alpha=1. Orange training curves, blue validation curves.

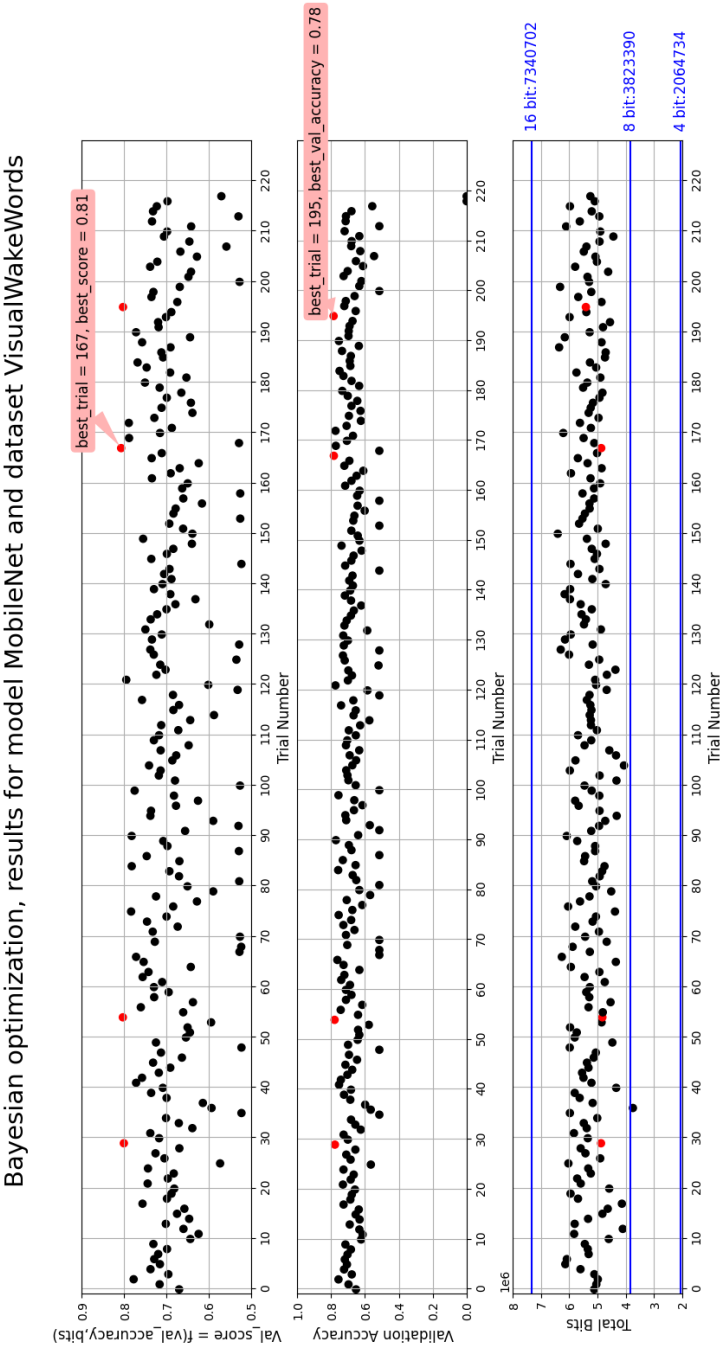


Figure 2.3: AutoQKeras search results for MobileNet on the Visual Wake Words dataset using $alpha = 1$. The points within 1% of the max val score (and the corresponding points in the validation accuracy and total bits graphs) are shown in red, while the blue horizontal lines show the total reference bits of the flat 16, 8 and 4-bit quantized models

Chapter 3

MLPerf Tiny Benchmark

3.1 Introduction

Tiny Machine Learning, also known as Tiny ML, is an emerging field that seeks to bring machine learning models to low-power electronic devices which are characterized by limited processing power and memory, such as embedded micro-controllers. In order to develop deep learning algorithms capable of running them, Tiny ML studies how techniques, like DNN quantization (Sec. 1.4), could be applied to meet these stringent constraints. Fueled by the growing market of IoT solutions, Tiny ML is the enabling technology behind the "smart" devices that can be found in many households, such as Amazon Alexa. But this technology isn't only limited to consumer electronics like voice assistants, for instance it is also used in the automotive industry for autonomous driving and in the medical field for diagnosing illnesses.
Overall, the transition from cloud-based machine learning to data processing on the edge offers several advantages:

- devices no longer require a constant connection to the internet, which leads to lower power consumption and increased responsiveness
- improved privacy and security, as data is processed on-device

As Tiny ML devices become increasingly widespread, being able to evaluate their performance becomes paramount. Therefore benchmarks were specifically designed for this purpose, facilitating developers in assessing the model and finding areas for improvement.
In view of this, we show how AutoQKeras (Sec. 2.3) can be used to quantize the four networks from the MLPerf Tiny Benchmark Suite [36], an open-source benchmark developed for Tiny ML applications. The suite consists of four benchmarks, tailored for a specific application scenario. Each benchmark consists in a dataset, a reference

model and a quality target, which are shown in Tab. 3.1 and will be described thoroughly in the next sections. The quality target is the minimum performance required to participate to the MLPerf Tiny divisions [36]. Moreover, the official benchmark repository on GitHub [2], provides training and testing scripts, pre-trained models and TFLite models.

Use Case	**Dataset**	**Model**	**Quality Target**
Visual Wake Words	VWW Dataset	MobileNetV1	80% (Top-1)
Image Classification	CIFAR10	ResNet	85%(Top-1)
Keyword Spotting	Speech Commands	DS-CNN	90% (Top-1)
Anomaly Detection	ToyADMOS	FC-AutoEncoder	0.85 (AUC)

Table 3.1: MLPerf Tiny Benchmark summary as in [36]

3.2 Project Outline

We will use AutoQKeras, customized as shown in Sec. 2.5, to find an optimal mixed-precision configuration for the four networks from the MLPerf Tiny Benchmark applying Bayesian Optimization. Additionally, in order to compare the quality of the mixed-precision solutions, QKeras is used to perform a flat 16, 8 and 4 bit quantization on all networks, i.e. all weight and activation layers are quantized with the same number of bits. The exploration of the quantization search space was carried out on Politecnico di Torino's Nvidia GTX 1070 GPU.

The main software libraries employed in this project are listed below:

- tensorflow 2.4.0
- keras 2.11
- qkeras 0.9.0 custom
- tensorboard 2.10.1 (visualization tool implemented as a callback for keras)
- keras-tuner 1.3.0 (hyperparameter optimization library)
- python 3.8.13
- librosa 0.9.2 (audio processing library)
- ffprobe 0.5 (multimedia stream analyzer)
- PyDub 0.25.1 (audio processing library)
- jupyter 1.0.0

The project is structured as follows:

- scripts: contains all the Python scripts that are used for the AutoQKeras search, flat quantization and inference of the MLPerf Tiny models;
- Py: general-use python scripts (e.g. split datasets, or extract test data);
- dataset: includes the full datasets for each benchmark and additional reduced datasets used for the quantization space exploration phase;
- ref_model: directory that consists of pre-trained model weight files which are provided by the official MLPerf Tiny repository on GitHub;
- flat_quantization: where flat quantization results are stored;
- results: where the AutoQKeras and re-trained best model results are stored;
- accelerators: contains the C++ implementation of the reconfigurable Conv2D [39], DepthwiseConv2D [38] and FC accelerators that are used in Sec. 4.

3.3 Modifications to the Keras model to implement affine quantization mapping

As discussed in Sec. 2.1 QKeras does not implement quantization on feature maps, but only on weights and biases. So each model from the MLPerf Tiny Benchmark, namely MobileNet, ResNet, DS-CNN, FC-AutoEncoder, has been modified to take this into account and correctly apply a uniform affine quantization mapping with the custom *quantized_bits_featuremap* class. To create the quantized QKeras models starting from the Keras models of the four networks, we use the method *model_quantize* which is automatically called by AutoQKeras on its input Keras model. The procedure is the following:

1. Apply a fake activation layer **before each** 2DConv, DWConv and FC layer. This Activation can be of any kind [1] as long as the user defines it in the Keras form Activation("name") and specifies in AutoQKeras' configuration dictionary that it should be replaced with a quantized_bits_featuremap quantization layer.

[1] If "name"="sigmoid" in Activation("name"), when AutoQKeras calculates the total number of bits, it will assign a constant number of bits determined by the output_bits parameter to all Activation("sigmoid") layers, regardless of the chosen quantizer. To correctly calculate the number of bits of the input model, ForgivingBits.py [11] has been modified. An alternative would be to avoid using Activation("sigmoid") and instead use Activation("relu"), but this could create confusion when reading the architecture of the Keras model

2. The **last** 2DConv, DWConv or FC layer must be **followed** by a fake Activation layer as described in point 1.

3. The model's original activation functions must be defined in the "direct" Keras form (i.e. ReLU() instead of Activation("relu")) to avoid being substituted with *quantized_bits_featuremap* by AutoQKeras.

4. **Every** Batch Normalization layer that can be folded with previous 2DConv, DWConv must be fused by enabling folding in the AutoQKeras class as described in Sec. 2.5

5. **Every** Average Pooling and Add layer must be enclosed by fake Activation layers

More details about this procedure can be found in the notebook called QKeras_Unveiled_Final.ipynb.

3.4 General structure of the quantization scripts

This section will go over a high level overview of the structure of the quantization scripts available in the *script* directory. Each script follows the same pattern:

1. Import the pre-trained MLPerf Tiny modified model, ensuring that the quantized model is able to recover most of its original accuracy after just a few epochs.

2. Define the parameters for the AutoQKeras Bayesian search: the quantization configuration, limit and goal dictionaries, which set the search space, maximum number of bits for quantization per layer and optimization goal (i.e. "energy" or "bits"), respectively. In this case we are interested in integer-only arithmetic, so we define the quantization search space as follows:

   - kernel: quantized_bits(x,x,1,1,alpha="auto")
   - bias: quantized_bits(y,y,1,1,alpha="auto")
   - activation: quantized_bits_featuremap(x,x,1,1,alpha="auto",scale_axis=0)

   where x can be 16, 8 or 4 and y can be 31 or 16. The limit Python dictionary is set as to ensure the whole quantization space can be explored, that is weights (kernels) and biases are quantized per-channel, while activations per-layer (scale_axis = 0), according to the theory in 2.1. The number of bits for input and weights has been chosen according to the supported precisions of the hardware accelerators (see Sec. 4), while the number of bits for the biases are kept higher because it has been proven that low bitwidths hurt accuracy substantially [24]. The goal is "bits" and its Python dictionary is the same as the preliminary research of Sec. 2.4.

3. Launch AutoQKeras passing the pre-trained and modified Keras model of point 1) and the dictionaries of point 2).

4. At the end of the AutoQKeras exploration, the best model for each benchmark is chosen according to its validation score (Eq. 2.5) and re-trained following the same settings defined in its own training script provided in the MLPerf Tiny repo [2], after importing the weights from the best epoch of the best trial.

5. This model is finally evaluated on the test set.

Additionally, each script is improved by adding the following Keras callbacks:

- early stop: to stop training of poor performing models in the AutoQKeras search
- csv logger: to store a csv log of the autoqkeras trials and best model re-training metrics
- model checkpoint: to save the weights of the best model for each epoch of re-training
- tensorboard: to generate TensorBoard [13] plots

Finally the script provides a section for training models with flat quantization (with 16, 8, and 4 bits for all layers) starting from the FP pre-trained weights, whose results are used for comparison against the best mixed-precision model. It is also possible to execute only a subset of the the previous steps by setting the appropriate flags at the top of the script.

3.5 Visual Wake Words

The first benchmark in the MLPerf Tiny Benchmark suite is the Visual Wake Words challenge and it is one of the two benchmarks, the other being Image Classification, that focuses on Tiny vision models. In particular, this use-case scenario targets person detection, a common vision task for low-cost imaging sensor applications that trigger signals when an individual is spotted. This has diverse applications that range from smart door bells to security cameras that can identify when a person is present and forward the image to the cloud for further processing.

The Visual Wake Words Dataset [5] is a collection of 109619 96×96 RGB images split into 53140 person images and 56479 non-person images. The dataset is directly obtained from the MSCOCO2014 dataset [29], pre-processed by assigning a label 1 to images containing a person with a minimal bounding box equal 2.5% and

label 0 otherwise. The preprocessing script, *buildPersonDetectionDatabase.py* can be found in the GitHub repository from Silicon Labs [26].

According to [36] for this application MobileNet reaches an accuracy of 86% across the preprocessed test set. However, it is not clear how or where this test set was obtained [40]. Moreover, the *evaluate* directory of the repository contains a file showing a list of 1000 images, presumably used for testing. These images are a subset of the ones found in the training/validation set, thus some of these have most likely been used to train the weights of the model that is provided by the repository, probably boosting the test set results. To verify if we could use those 1000 images as test set we run inference obtaining 85.1% accuracy that is close to what is declared in [36]. It is possible to obtain 86% accuracy by training this model for 10 more epochs with respect to the 50 epochs of the MLPerf Tiny training script. Admittedly, this is not a standard ML procedure, as by definition data from the test set should never be used during training.

Another solution that was attempted was to retrain the model on the full training/validation set by removing, a priori, the 1000 image subset. This approach however provides inconsistent results with what is declared by the paper, since inference on this new model returns an 82% test accuracy. The results of these two experiments led us to choose the first approach, i.e. using those 1000 images as test set for VWW.

The model used for this scenario is MobileNetV1 [17], which was already mentioned in Sec. 1.3.1 and was modified according to the principles illustrated in Sec. 3.3. MLPerf Tiny trains the model on the full VWW dataset, using data

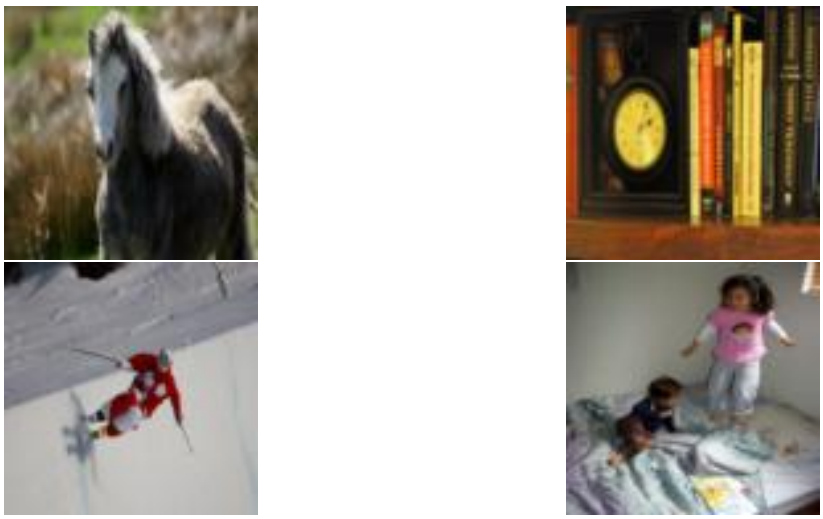


Figure 3.1: Four sample images, two person and two non-person, from the VWW Dataset [5]

augmentation and the following settings:

- optimizer: Adam
- loss: categorical crossentropy
- metrics: accuracy
- epochs: 50
- batch size: 32
- variable learning rate: 0.001 for the first 20 epochs, 0.0005 for the next 10 epochs, and 0.00025 until the end
- train/val split: 0.1

The resulting weights can be found in the official MLPerf Tiny repository as *vww_96.h5* and are used in the next phase: the AutoQKeras quantization space exploration. During this phase, the dataset is reduced by 75% choosing random images from both classes to speedup the training of each trial and so the entire AutoQKeras research time. Moreover, the research started from the pre-trained FP weights to help the model recover faster from the loss in accuracy due to quantization. Since a pre-trained model is being used, the learning rate is fixed at 0.00025, the value used for the last epochs of the original training procedure. In addition, data augmentation is disabled to provide comparable score values among trials. A total of 210 trials, 8 epochs maximum each (if not stopped by early-stopping), were run for this network. The results are shown in Fig. 3.2, that presents the validation score (Eq. 2.5), validation accuracy and total bit (Eq. 2.4) values for each trial. The flat quantization reference sizes were obtained by quantizing weights and activations to 16, 8 and 4 bits while maintaining a constant 31 bit quantization for biases.

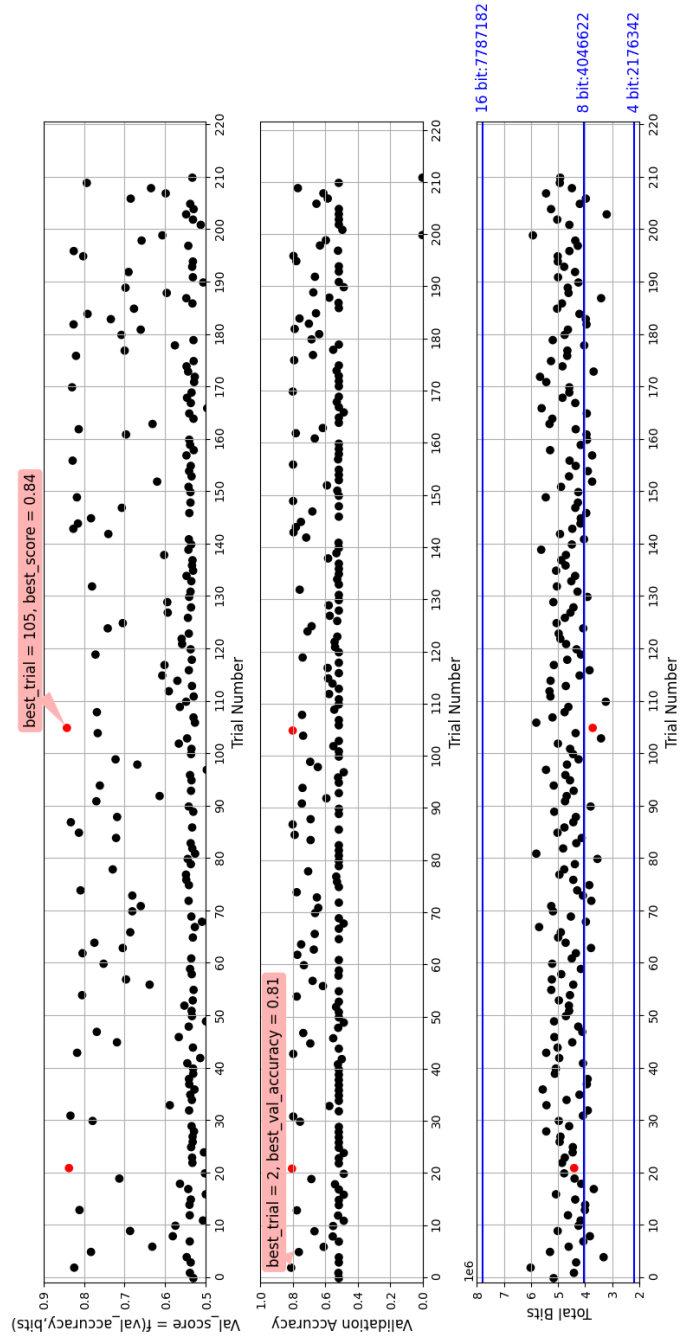
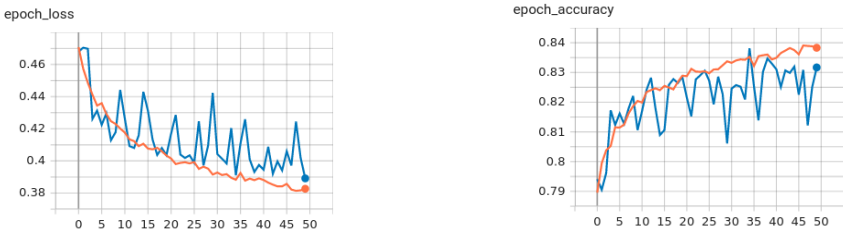


Figure 3.2: AutoQKeras search results for MobileNet on the VWW dataset. The points within 1% of the max val score (and the corresponding points in the validation accuracy and total bits graphs) are shown in red, while the blue horizontal lines show the total reference bits of flat 16, 8 and 4 quantized models

By looking at the graph (Fig. 3.2) the model appears to be particularly difficult to quantize as there are multiple jumps between validation score points around 0.5 and 0.8 and there is no improving trend with the passing of trials but only a random one. In this case it is very likely that Bayesian Optimization has not converged to an optimal value and definitely requires more trials to find better models. The best model obtained by the research is at trial 105, with a validation score equal to 0.84 and a validation accuracy equal to 0.81, while the bit size is reduced by 52.25% and 8.10% with respect to the 16 bit flat model (7787182 to 3718680) and the 8 bit flat model (4046622 to 3718680).

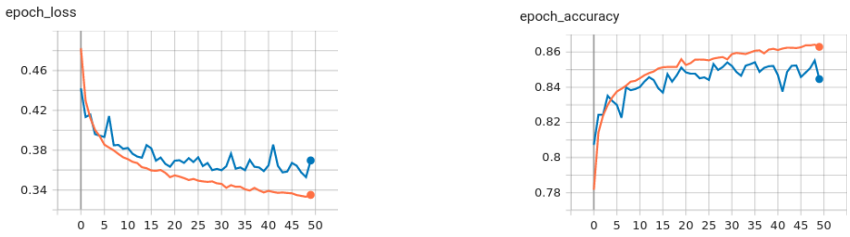
This model was then re-trained, this time on the full dataset with augmentation, and the following training/validation curves are obtained(Fig. 3.3).



(a) Loss **(b)** Accuracy

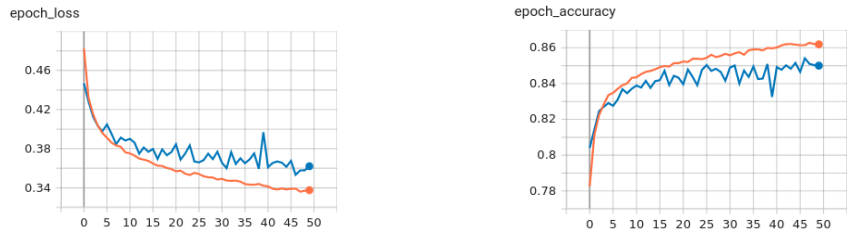
Figure 3.3: Best mixed-precision MobileNet training (orange) and validation (blue) plot

Comparing this result to the model implemented in Sec. 2.6 it is clear that using an affine quantization with scale and zeropoint provides better results in terms of bit reduction being 23.58% smaller (3718684 vs 4866150) with a negligible degradation in accuracy of only 0.8% (85.00% vs 85.79%). Moreover, the model loses just 1% accuracy with respect to the FP model. Considering that the training curves of Fig. 3.3 show a good training (no overfitting, no underfitting) and monotonic trends for both loss (a) and accuracy (b), the training could have also been continued for some more epochs, which could have probably closed the gap in accuracy. However we stopped training at 50 epochs to have a fair comparison with the results of Sec. 2.6 Fig. 2.1.



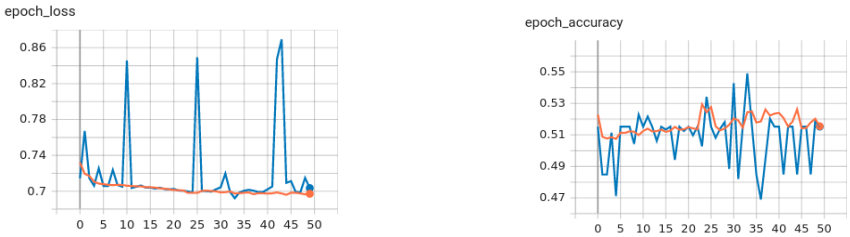
(a) Loss **(b)** Accuracy

Figure 3.4: Training (orange) and validation (blue) plots of MobileNet quantized with 16-bit flat



(a) Loss **(b)** Accuracy

Figure 3.5: Training (orange) and validation (blue) plots of MobileNet quantized with 8-bit flat



(a) Loss **(b)** Accuracy

Figure 3.6: Training (orange) and validation (blue) plots of MobileNet quantized with 4-bit flat

A recap of the results for the best quantized models for VWW is shown in Tab. 3.2

Quantization Policy	**Accuracy** [%]	**Error** [%]	**Total Bits**	**Tot. Bit Reduction** [%]	**Best Epoch**
FP	86.00				
Flat 16	87.30	-1.30	7787182		47
Flat 8	87.19	-1.19	4046622	48.03	47
Flat 4	50.00	17.51	2176342	72.05	33
MP	85.79	0.21	4866150	37.51	49
MP Affine	85.00	1.00	3718680	52.25	39

Table 3.2: Comparison between the best mixed-precision and flat quantized MobileNet models on the VWW 1000 image test set. Error w.r.t. FP values and Bit Reduction w.r.t. Flat 16. Minimum acceptable accuracy for submission to MLPerf Tiny contest: 0.80

For what concerns the flat quantized models, as expected, after re-training the 16-bit and 8-bit flat models are able to recover the accuracy of the FP model, while the 4-bit model cannot be trained successfully. The models were evaluated on the test set obtaining 87.30%, 87.19%, 50.00% accuracy, respectively. Finally, the bitwidth configuration of the best mixed-precision model is shown in Fig. 3.7

```
stats: delta_p=0.05 delta_n=0.05 rate=2.0 trial_size=3718680 reference_size=7787182
       delta=5.33%
       a_bits=2507784/4372512 (-42.65%) p_bits=1210896/3414670 (-64.54%)
       total=3718680/7787182 (-52.25%)
activation           quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
conv2d               f=8 quantized_bits(4,4,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_1         quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
depthwise_conv2d     f=None quantized_bits(16,16,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_2         quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
conv2d_1             f=16 quantized_bits(16,16,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_3         quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
depthwise_conv2d_1   f=None quantized_bits(8,8,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_4         quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
conv2d_2             f=32 quantized_bits(16,16,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_5         quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
depthwise_conv2d_2   f=None quantized_bits(4,4,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_6         quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
conv2d_3             f=32 quantized_bits(4,4,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_7         quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
depthwise_conv2d_3   f=None quantized_bits(4,4,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_8         quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
conv2d_4             f=64 quantized_bits(8,8,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_9         quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
depthwise_conv2d_4   f=None quantized_bits(8,8,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_10        quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
conv2d_5             f=64 quantized_bits(8,8,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_11        quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
depthwise_conv2d_5   f=None quantized_bits(8,8,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_12        quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
conv2d_6             f=128 quantized_bits(16,16,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_13        quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
depthwise_conv2d_6   f=None quantized_bits(8,8,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_14        quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
conv2d_7             f=128 quantized_bits(4,4,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_15        quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
depthwise_conv2d_7   f=None quantized_bits(8,8,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_16        quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
conv2d_8             f=128 quantized_bits(8,8,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_17        quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
depthwise_conv2d_8   f=None quantized_bits(16,16,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_18        quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
conv2d_9             f=128 quantized_bits(4,4,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_19        quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
depthwise_conv2d_9   f=None quantized_bits(16,16,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_20        quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
conv2d_10            f=128 quantized_bits(4,4,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_21        quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
depthwise_conv2d_10  f=None quantized_bits(8,8,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_22        quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
conv2d_11            f=128 quantized_bits(8,8,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_23        quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
depthwise_conv2d_11  f=None quantized_bits(8,8,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_24        quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
conv2d_12            f=256 quantized_bits(4,4,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_25        quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
depthwise_conv2d_12  f=None quantized_bits(4,4,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_26        quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
conv2d_13            f=256 quantized_bits(4,4,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_27        quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
activation_28        quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
dense                u=2 quantized_bits(4,4,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_29        quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
```

Figure 3.7: Best mixed-precision MobileNet configuration found by AutoQKeras (Trial 105 in Fig. 3.2)

3.6 Image Classification

Image Classification is the second MLPerf Tiny benchmark that involves a computer vision application. This task consists in training a machine to extract features from various images and to classify them in predetermined categories, which is common in applications such as self-driving cars that need to recognize objects in the environment such as other vehicles, pedestrians and road signs. The benchmark uses the CIFAR-10 dataset [25], which consists of 60000 32×32 RGB images belonging to 10 mutually-exclusive classes, 6000 each (Fig. 3.8).

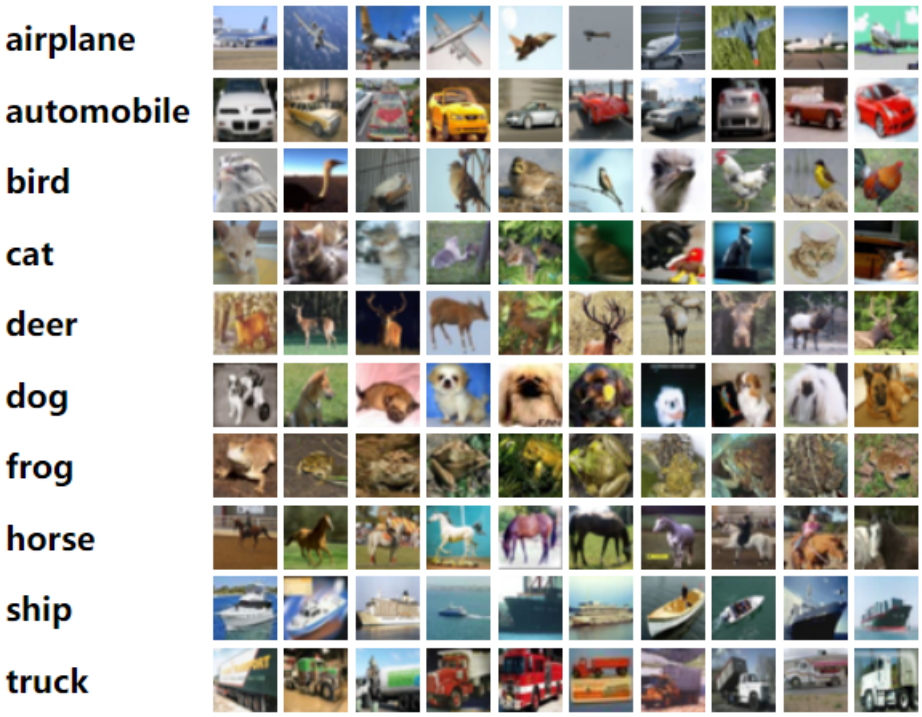


Figure 3.8: 10 random images from each category in the CIFAR-10 dataset [25]

The dataset is provided as a "pickled" binary dataset that consists of 5 batches containing 10000 images each, with a training/validation split of 0.2. The remaining 10000 images are set aside for the test set. MLPerf Tiny further reduces this test set to a performance evaluation dataset (or *perf*[2]) of 200 images, which can be obtained by enabling the appropriate flags inside the test script from the benchmark repository.

[2]the perf tests are the ones specified by MLPerf Tiny in the evaluation section of their repo

The original training script uses a custom ResNet model (see Sec. 1.3.2) trained with the following parameters:

- optimizer: Adam
- loss: categorical crossentropy
- metrics: accuracy
- epochs: 500
- batch size: 32
- decaying learning rate given by $0.001 \times 0.99^{epoch}$

The floating point model is trained using augmented data and reaches 87 % accuracy on the performance set.

To run the AutoQKeras exploration phase, the dataset was reduced by 40% taking the first 3 batches of pickled training/validation data. Again, each trial consisted in a maximum of 8 epochs where the data generator was disabled and the learning rate set was set to $lr = 0.00002967$, an intermediate value with respect to the range spanned by the lr across the 500 training epochs, to avoid choosing extremely small values (e.g. when epoch=500 then lr = 6.57e-6). As shown in Fig. 3.9 the Bayesian Optimization shows an improving trend that stabilizes after trial 170.

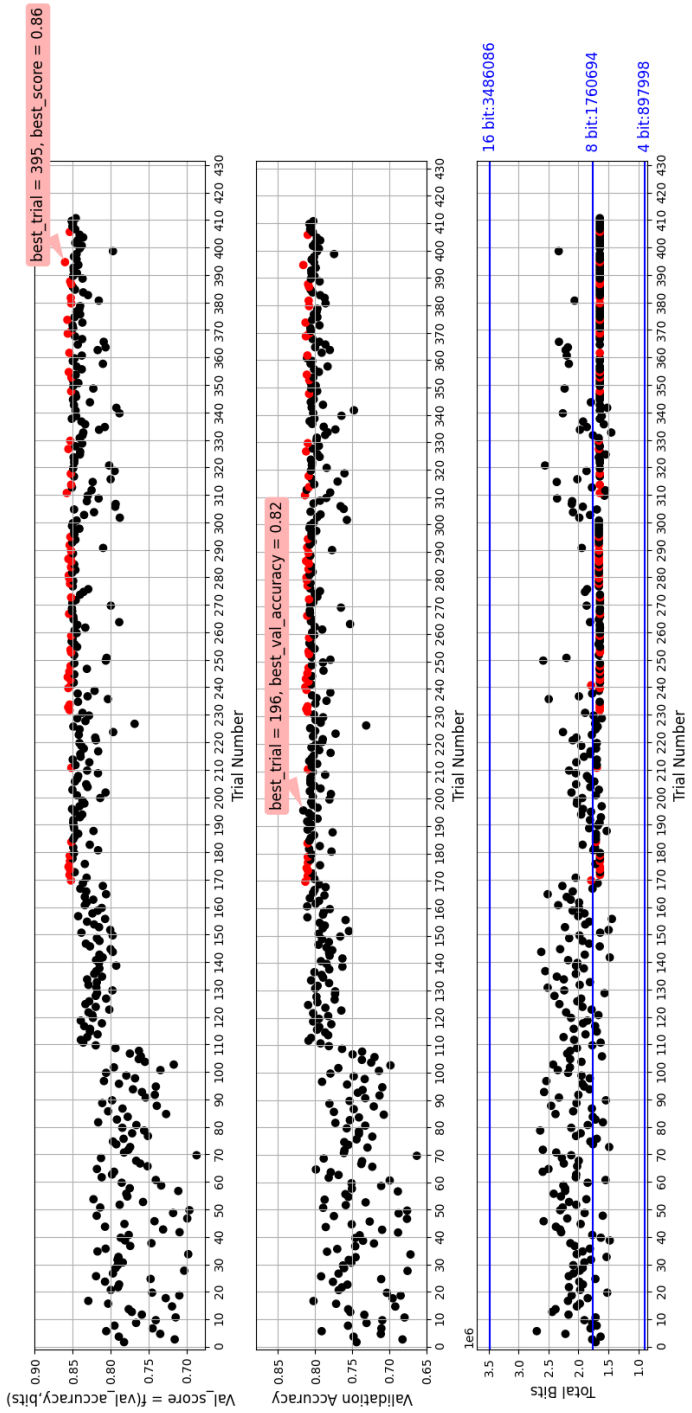


Figure 3.9: AutoQKeras search results for ResNet on the CIFAR-10 dataset. The points within 1% of the max val score (and the corresponding points in the validation accuracy and total bits graphs) are shown in red, while the blue horizontal lines show the total reference bits of flat 16,8 and 4 quantized models

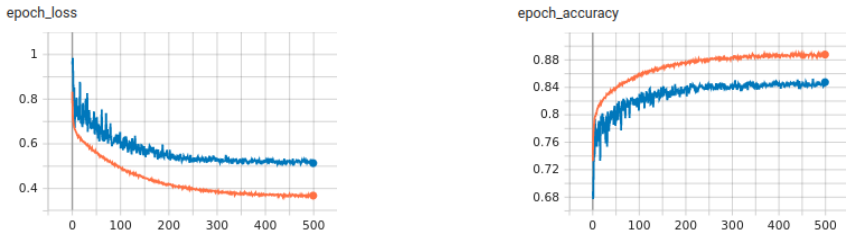
The best model is found at trial 395 and reaches 87.50% accuracy on the MLPerfTiny performance set after re-training [3] (Fig. 3.11), recovering (and even surpassing) the accuracy of the FP model (87.00%). This phenomenon could be explained with the small size and simplicity of the perf set and, possibly, the fact that MLPerf Tiny uses a different validation split from ours. In fact, when using the full CIFAR-10 test set, the FP performs better than the MP model (87.19% vs 84.07%). Also, MLPerf Tiny uses data augmentation to train the FP model, but does not provide a seed: after re-training the FP model without importing the pre-trained weights we obtain 88.49% accuracy on the perf test set. The model's quantization configuration is shown in Fig. 3.10.

```
stats: delta_p=0.05 delta_n=0.05 rate=2.0 trial_size=1641510 reference_size=3486086
       delta=5.43%
       a_bits=1139872/2237600 (-49.06%) p_bits=501638/1248486 (-59.82%)
       total=1641510/3486086 (-52.91%)
activation          quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
conv2d              f=16 quantized_bits(16,16,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_1        quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
conv2d_1            f=16 quantized_bits(8,8,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_2        quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
conv2d_2            f=16 quantized_bits(16,16,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_3        quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
activation_4        quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
conv2d_3            f=32 quantized_bits(8,8,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_5        quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
activation_7        quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
conv2d_4            f=32 quantized_bits(8,8,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
conv2d_5            f=32 quantized_bits(8,8,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_6        quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
activation_8        quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
activation_9        quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
conv2d_6            f=64 quantized_bits(8,8,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_10       quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
activation_12       quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
conv2d_7            f=64 quantized_bits(4,4,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
conv2d_8            f=64 quantized_bits(8,8,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_11       quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
activation_13       quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
activation_14       quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
dense               u=10 quantized_bits(8,8,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_15       quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
```

Figure 3.10: Best mixed-precision ResNet configuration found by AutoQKeras (Trial 395 in Fig. 3.9)

As the MP model, the 16- and 8-bit flat models, evaluated on the MLPerf Tiny perf set, recover and surpass the FP accuracy reaching 89.49% and 88.99%, respectively. Instead, when evaluated on the full CIFAR-10 test set, these models both lose some accuracy, with the 16-bit flat model reaching 84.05% and the 8-bit model reaching 84.18%. As usual, the 4-bit flat model is not able to recover the original FP accuracy, degrading it by 9.01% on the perf set and by 13.68% on the full test set.

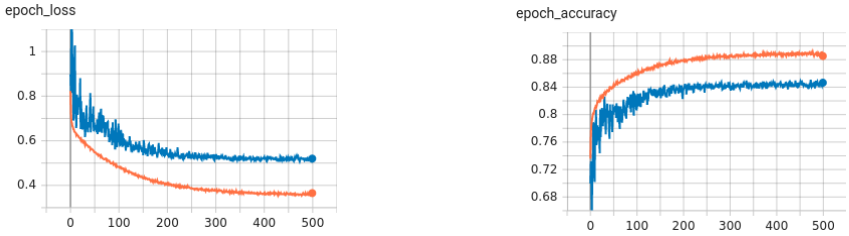
[3]The best MP model was re-trained from scratch, following the same training configuration specified by the train script provided by MLPerf Tiny. In all other cases, the models were re-trained starting from the AutoQKeras checkpoint.



(a) Loss **(b)** Accuracy

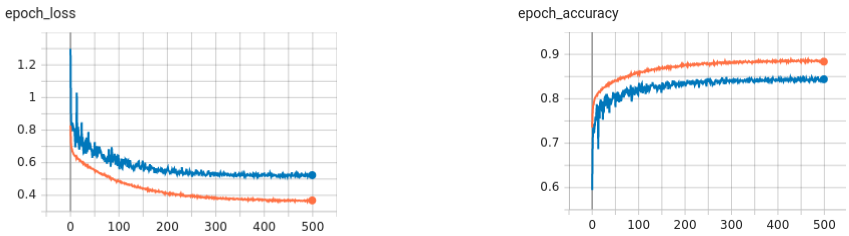
Figure 3.11: Best mixed-precision ResNet training (orange) and validation(blue) plot

While the flat 16- and 8-bit models provide a slightly better accuracy with respect to the best mixed model (+1.99% and +1.49%), the latter provides a 52.91% (1641510 vs 3486086) and a 6.77% (1641510 vs 1760694) reduction in terms of total bits, respectively.



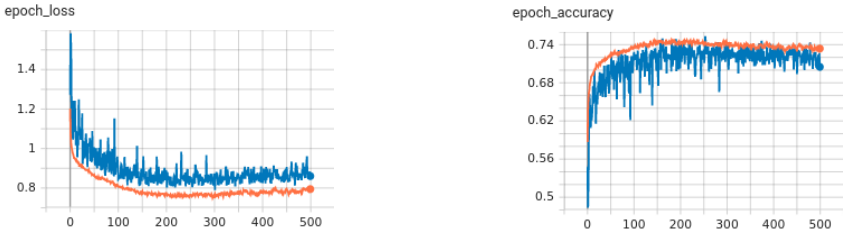
(a) Loss **(b)** Accuracy

Figure 3.12: Training (orange) and validation (blue) plots of ResNet quantized with 16-bit flat



(a) Loss **(b)** Accuracy

Figure 3.13: Training (orange) and validation (blue) plots of ResNet quantized with 8-bit flat



(a) Loss **(b)** Accuracy

Figure 3.14: Training (orange) and validation (blue) plots of ResNet quantized with 4-bit flat

Quantization Policy	**Accuracy** [%]	**Error** [%]	**Total Bits**	**Tot. Bit Reduction** [%]	**Best Epoch**
FP	87.00*				
Flat 16	89.49	-2.49	3486086		500
Flat 8	88.99	-1.99	1760694	49.49	500
Flat 4	77.99	9.01	897998	74.24	250
MP	87.50	-0.50	1641510	52.91	500

Table 3.3: Comparison between the best mixed-precision and flat quantized ResNet models on the CIFAR-10 perf set (200 samples). Error w.r.t. FP values and Bit Reduction w.r.t. Flat 16. Minimum acceptable accuracy for submission to MLPerf Tiny contest: 0.85. (*We obtain 88.49 (epoch 495) re-training the FP model, instead of importing MLPerf Tiny's pre-trained weights)

Quantization Policy	**Accuracy** [%]	**Error** [%]	**Total Bits**	**Tot. Bit Reduction** [%]	**Best Epoch**
FP	85.92*				
Flat 16	84.05	1.87	3486086		500
Flat 8	84.18	1.74	1760694	49.49	500
Flat 4	76.27	9.65	897998	74.24	250
MP	84.07	1.85	1641510	52.91	500

Table 3.4: Comparison between the best mixed-precision and flat quantized ResNet models on the full CIFAR-10 test set (10000 samples). Error w.r.t. FP values and Bit Reduction w.r.t. Flat 16. Minimum acceptable accuracy for submission to MLPerf Tiny contest: 0.85. (*FP model re-trained from random weights instead of importing MLPerf Tiny's pre-trained weights)

3.7 Keyword spotting

The Keyword Spotting benchmark is used to evaluate the performance of a specific category of speech processing ML applications that try to detect a predetermined set of keywords from audio samples. This is the technology behind very popular virtual assistants like Google Assistant or Amazon Alexa. This benchmark uses the Speech Commands V2 dataset [42], a collection of 105,829 utterances collected from 2,618 speakers with a variety of accents. This dataset, which contains 30 words and a set of background noises, is divided into standard training, validation and test splits, ensuring that any particular speaker only appears in one subset. The dataset is further preprocessed for this benchmark by splitting the 30 words into a set of 10 words plus two classes called "silence" and "unknown". The latter is obtained by combining the background noises with the remaining 20 words. To sum up, the final dataset used by MLPerf Tiny is made of the following 10 words and 2 classes: down, go, left, no, off, on, right, stop, up, yes, and the classes silence and unknown. By default the audio feature representation used in this benchmark is Mel-Frequency Cepstral Coefficients (MFCC), a technique that converts audio signals to the frequency domain (Mel Spectrogram) using the Mel scale, which is a non-linear frequency perceptual scale of pitches judged by listeners to be equal in distance from one another. The Mel spectrogram is then transformed using a Discrete Cosine Transform (DCT) to obtain the MFCCs which can be used to train the neural network. The dataset is provided as a Tensorflow dataset which is automatically downloaded and split by the *tfds.load* method and imported as a *tf.data.Dataset* object. The benchmark's target network is a Depthwise Separable Convolutional Neural Network (DS-CNN) which, as the name suggests, uses depthwise separable convolutions. The floating-point model is able to reach 92.3% accuracy on the test set, after being trained with the following parameters:

- optimizer: Adam
- loss: Sparse Categorical Crossentropy
- metrics: accuracy
- epochs: 36
- batch size: 100
- step learning rate: 0.0005 for the first 12 epochs, 0.0001 for the next 12 and finally 0.00002
- shuffle input
- train/val split = 0.1

The AutoQKeras exploration phase follows the same logic as the previous benchmarks. In particular the dataset is reduced by 75% and the model trained for 8 epochs maximum per trial (depending on early-stopping) with a fixed learning rate of 0.001 to speed up learning. The best model was found at trial 50, and the search plot (Fig. 3.16) shows that this is an easy network to quantize as most of the models show a values above 0.90 for both validation score and validation accuracy while maintaining the total bit count below the flat 8-bit line. The configuration for this model is shown in Fig. 3.15.

```
stats: delta_p=0.05 delta_n=0.05 rate=2.0 trial_size=574820 reference_size=1535460
       delta=7.09%
       a_bits=428976/1164976 (-63.18%) p_bits=145844/370484 (-60.63%)
       total=574820/1535460 (-62.56%)
activation           quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
conv2d               f=64 quantized_bits(16,16,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_1         quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
depthwise_conv2d     f=None quantized_bits(8,8,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_2         quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
conv2d_1             f=64 quantized_bits(4,4,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_3         quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
depthwise_conv2d_1   f=None quantized_bits(8,8,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_4         quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
conv2d_2             f=64 quantized_bits(4,4,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_5         quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
depthwise_conv2d_2   f=None quantized_bits(4,4,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_6         quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
conv2d_3             f=64 quantized_bits(4,4,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_7         quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
depthwise_conv2d_3   f=None quantized_bits(16,16,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_8         quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
conv2d_4             f=64 quantized_bits(4,4,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_9         quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
activation_10        quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
dense                u=12 quantized_bits(8,8,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_11        quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
```

Figure 3.15: Best mixed-precision DS-CNN configuration found by AutoQKeras (Trial 50 in Fig. 3.9)

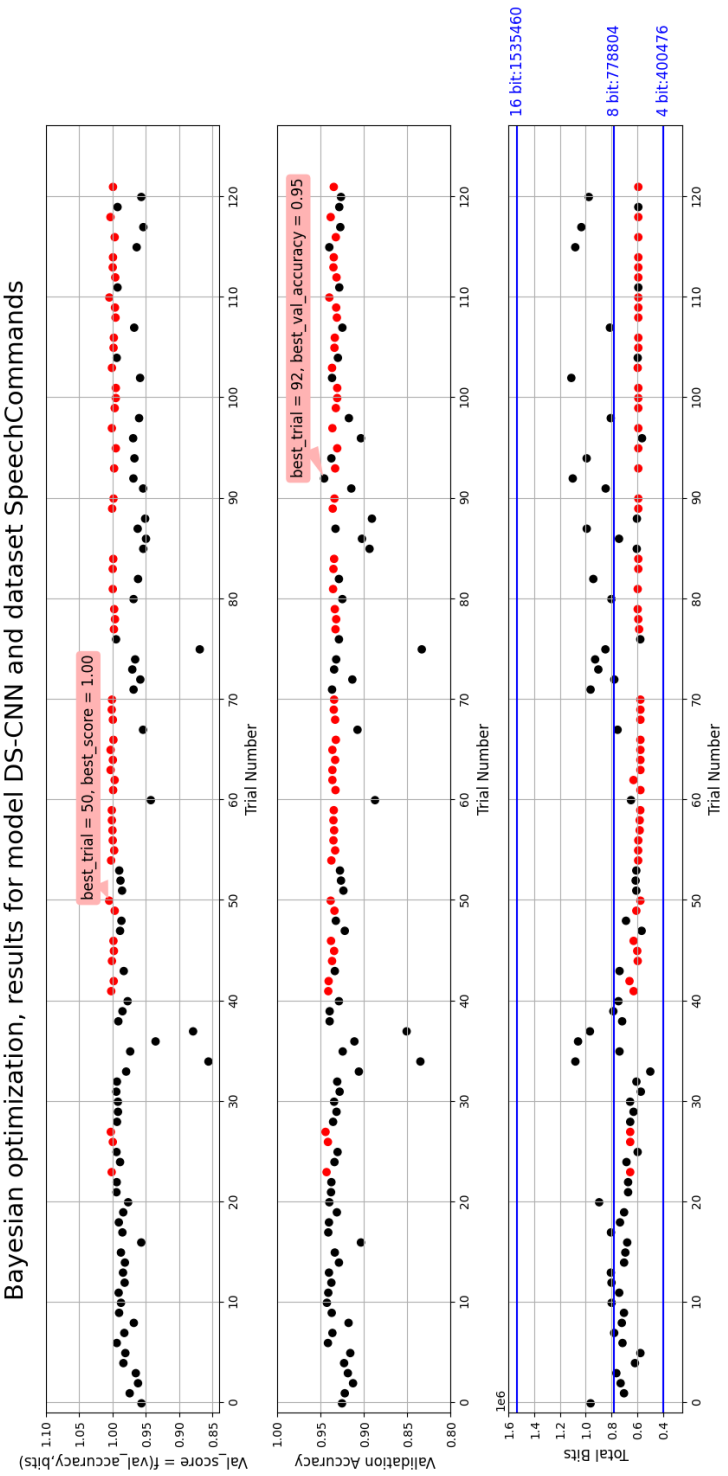
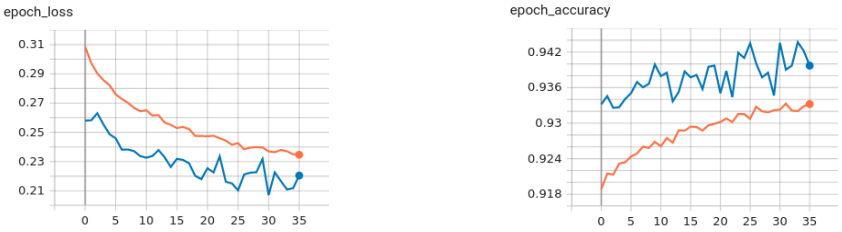


Figure 3.16: AutoQKeras search results for DS-CNN on the Speech Commands dataset. The points within 1% of the max val score (and the corresponding points in the validation accuracy and total bits graphs) are shown in red, while the blue horizontal lines show the total reference bits of flat 16, 8 and 4 quantized models

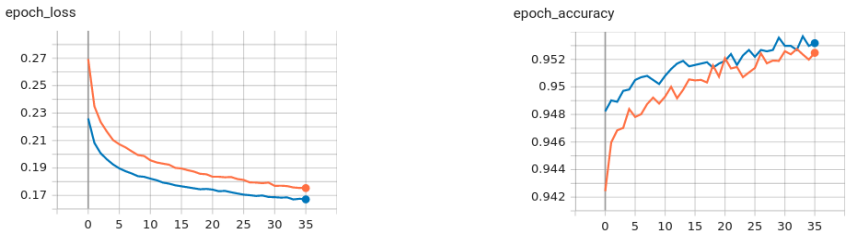
After re-training (Fig. 3.17), the best mixed-precision model is able to reach 90.47% accuracy on the test set, a 1.83% drop with respect to the floating point model.



(a) Loss **(b)** Accuracy

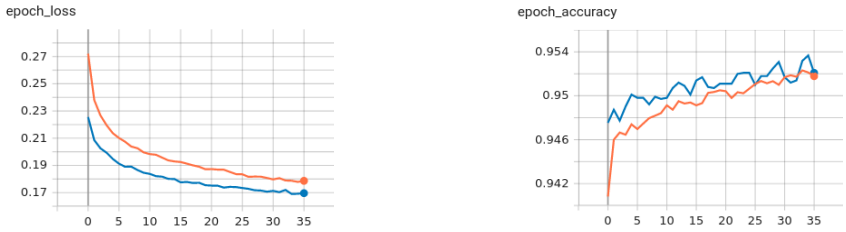
Figure 3.17: Best mixed-precision DS-CNN training (orange) and validation(blue) plot

The inference results for the best mixed and flat quantized models on the Speech Commands test set are shown in Tab. 3.5. As before, the mixed precision model shows a slight performance drop with respect 16- and 8-bit flat quantizations (-1.02% and -1.06%), however, it has a noticeably lower memory footprint (-62.56% and -26.2%).



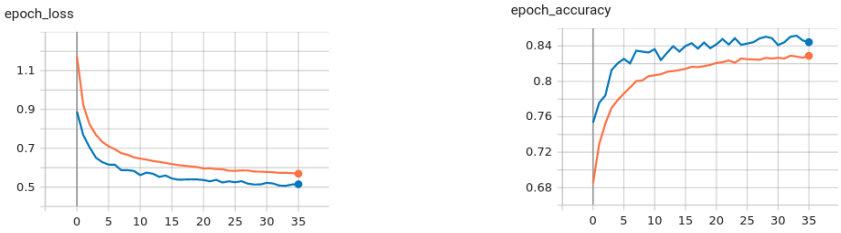
(a) Loss **(b)** Accuracy

Figure 3.18: Flat 16 bit quantized DS-CNN training (orange) and validation(blue) plot



(a) Loss **(b)** Accuracy

Figure 3.19: Flat 8 bit quantized DS-CNN training (orange) and validation(blue) plot



(a) Loss **(b)** Accuracy

Figure 3.20: Flat 4 bit quantized DS-CNN training (orange) and validation(blue) plot

Quantization Policy	**Accuracy** [%]	**Error** [%]	**Total Bits**	**Tot. Bit Reduction** [%]	**Best Epoch**
FP	92.30				
Flat 16	91.49	0.81	1535460		35
Flat 8	91.53	0.77	778804	49.27	35
Flat 4	73.82	18.48	400476	73.91	36
MP	90.47	1.83	574820	62.56	30

Table 3.5: Comparison between the best mixed-precision and flat quantized DS-CNN models on the Speech Commands V2 test set. Error w.r.t. FP values and Bit Reduction w.r.t. Flat 16. Minimum acceptable accuracy for submission to MLPerf Tiny contest: 0.90

3.8 Anomaly Detection

The last benchmark in the MLPerf Tiny suite is Anomaly Detection, a technique that was mentioned in Sec. 1.3.3 where the principles of the FC-AutoEncoder network were illustrated. This unsupervised learning task aims to detect machine failures at an early stage, which is a typical use case. These can be detected in various ways, for example monitoring temperature and/or vibrations but also by analyzing audio samples captured by microphones, which is the technique used to build the training/test dataset for this use-case. The dataset is taken from the DCASE2020 competition [22], which contains data from the ToyADMOS [23], and the MIMII dataset [35]. The dataset contains single-channel 10 second length audio samples from 6 different machine types mixed with environmental noise: slide rail, fan, pump, valve, toy-car, toy-conveyor. Each type in turn is made of 4 machine IDs. There are approximately 1000 samples per machine ID, for a total of 4000 samples per type. Due to this complexity, MLPerf Tiny benchmark considers only the ToyCar machine.
The dataset consists of a development dataset, an additional training dataset, and an evaluation dataset:

- development: training data belonging to the normal class only, divided among 4 machine IDs (4000 samples), plus a labeled test set containing data from both the normal and anomaly class (2459 samples)
- additional training dataset: training data belonging to the normal class only, for 3 new machine IDs (3000 samples)
- evaluation dataset: unlabeled data used by the DCASE2020 organizers for testing. We will not use this data since MLPerf Tiny evaluates the model using the performance set which will be explained later on.

As this is an audio dataset, it preprocessed in a similar fashion to the Speech Commands dataset using the MFCC technique. The features are then used to train the FC-AutoEncoder using these training configurations:

- optimizer: Adam
- loss: mean squared error
- metrics: Area Under the Curve (AUC) and Partial Area Under the Curve (pAUC)
- epochs: 100
- batch size: 512

- shuffle data
- train/val split: 0.9/0.1

Bearing in mind that this is an unsupervised learning problem, training consists in using only the normal data from the 7 machine IDs of the development dataset (i.e. the main development dataset plus the additional training dataset). After training, the FC-AutoEncoder should be able to reconstruct the input tensor with a minimum error as discussed Sec. 1.3.3. In contrast, testing uses both anomaly and normal data of 4 machine IDs to evaluate the FC-AutoEncoder. Each machine ID is treated as a separate entity: for each file of each ID an "anomaly score" is computed and used to calculate the AUC and pAUC metrics as shown in Eqs. 3.1. These metrics are used to evaluate binary classifier systems such as this one, measuring the area under the Receiver Operating Characteristic (ROC) curve, which is a plot of the True Positive Rate (TPR) against the False Positive Rate (FPR) for different classifier thresholds. An AUC equal to 1 defines a perfect classifier, that is able to completely distinguish the inputs. The pAUC is a metric which is computed restricting the range of FPR (or TPR) to regions which are deemed more critical, in this case the max FPR is 0.1.

$$\begin{aligned} error &= MSE(input - prediction) \\ anomaly\_score &= mean(error) \\ AUC &= f(y\_true, anomaly\_score) \end{aligned} \tag{3.1}$$

The floating point model evaluated on the perf test set achieves:

- average AUC = 88.70%
- average pAUC = 77.28%

Integrating an unsupervised learning problem in AutoQKeras cannot be done directly. The reason is that AutoQKeras needs a metric which is bounded between 0 and 1 (usually the validation accuracy) to compute the Forgiving Factor which is in turn needed for the validation score. However, since we train the model using samples from the normal class only, we do not have the validation accuracy metric. In this case the only significant metric that we have is the MSE loss calculated between input and prediction (Eqs. 3.1). Thus we define a new custom metric inversely proportional to the loss, which is bounded between 0 and 1, as reported in Eq. 3.2:

$$custom\_metric = \frac{1}{1 + \frac{loss}{10}} \tag{3.2}$$

At this point AutoQKeras is able to use this metric to compute a meaningful reward (Forgiving Factor) for the Bayesian Optimization oracle.

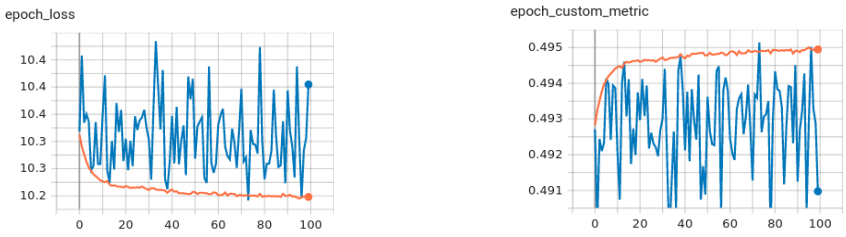
During an AutoQKeras trial the FC-AutoEncoder is trained for 8 epochs on the full training dataset using the training configuration provided in the original training script. It should be noted that in this specific case the Batch Normalization layers are not folded into the Dense layers as this feature is not currently implemented in QKeras. The hyperparameter exploration reported in Fig. 3.24 finds a mixed precision model at trial 167 that reduces the total memory footprint by 70.85% and 42.46% with respect to the flat 16- and 8-bit quantized models respectively[4].

The best mixed-precision configuration for FC-AutoEncoder is reported in Fig. 3.21. After re-training both flat (Figs. 3.23a,b,c)) and mixed-precision models (Fig. 3.22) (NOTE: The plots have different Y-axis ranges) following the same training procedure as MLPerf Tiny, we run inference on the ToyCar ToyADMOS and MIMII 248 sample perf dataset obtaining the results shown in Tab. 3.6.

```
stats: delta_p=0.05 delta_n=0.05 rate=2.0 trial_size=1259512 reference_size=4321016
       delta=8.89%
       a_bits=26752/42112 (-36.47%) p_bits=1232760/4278904 (-71.19%)
       total=1259512/4321016 (-70.85%)
activation          quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
dense               u=128 quantized_bits(4,4,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_1        quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
dense_1             u=128 quantized_bits(4,4,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_2        quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
dense_2             u=128 quantized_bits(4,4,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_3        quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
dense_3             u=128 quantized_bits(4,4,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_4        quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
dense_4             u=8 quantized_bits(4,4,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_5        quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
dense_5             u=128 quantized_bits(16,16,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_6        quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
dense_6             u=128 quantized_bits(4,4,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_7        quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
dense_7             u=128 quantized_bits(8,8,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_8        quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
dense_8             u=128 quantized_bits(8,8,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_9        quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
dense_9             u=640 quantized_bits(4,4,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_10       quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
```

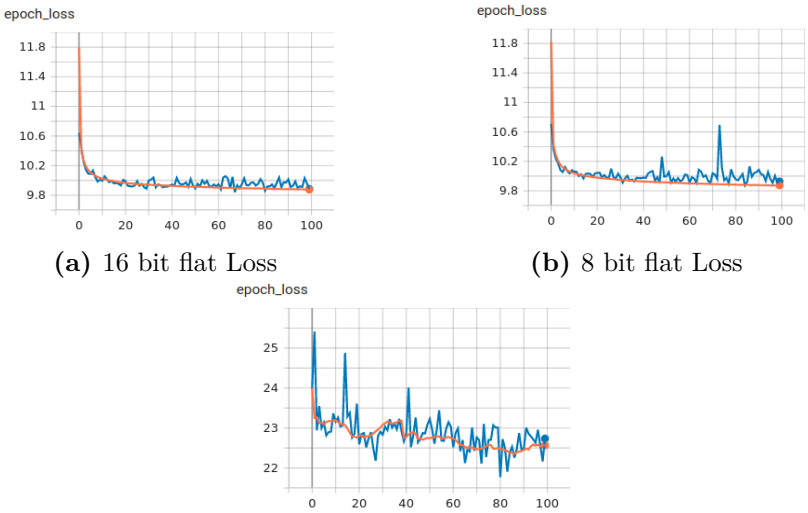
Figure 3.21: Best mixed-precision FC-AutoEncoder configuration found by AutoQKeras (Trial 50 in Fig. 3.24)

[4]The total bit count does not consider batch normalization parameters, which for the other models were folded



(a) Loss **(b)** Custom Metric 3.2

Figure 3.22: Best mixed-precision FC-AutoEncoder training (orange) and validation(blue) plot. Notice how the custom metric is inversely proportional to the loss metric



(a) 16 bit flat Loss **(b)** 8 bit flat Loss

(c) 4 bit flat Loss

Figure 3.23: Flat 16 (a), 8 (b) and 4-bit (c) quantized FC-AutoEncoder training (orange) and validation(blue) Loss plot

Quant. Policy	**AUC** [%]	**Error** [%]	**pAUC** [%]	**Error** [%]	**Total Bits**	**Tot. Bit Red.** [%]	**Best Epoch**
FP	88.70		77.28				
Flat 16	86.73	1.97	75.1	2.18	4321016		80
Flat 8	88.58	0.12	77.05	0.23	2188984	49.34	85
Flat 4	76.72	11.98	65.14	12.14	1122968	74.01	81
MP	87.50	1.20	76.57	0.71	1259512	70.85	97

Table 3.6: Comparison between the best mixed-precision and flat quantized FC-AutoEncoder models on ToyCar ToyADMOS and MIMII 248 sample perf dataset. Error w.r.t. FP values and Bit Reduction w.r.t. Flat 16. Minimum acceptable AUC for submission to MLPerf Tiny contest: 0.85

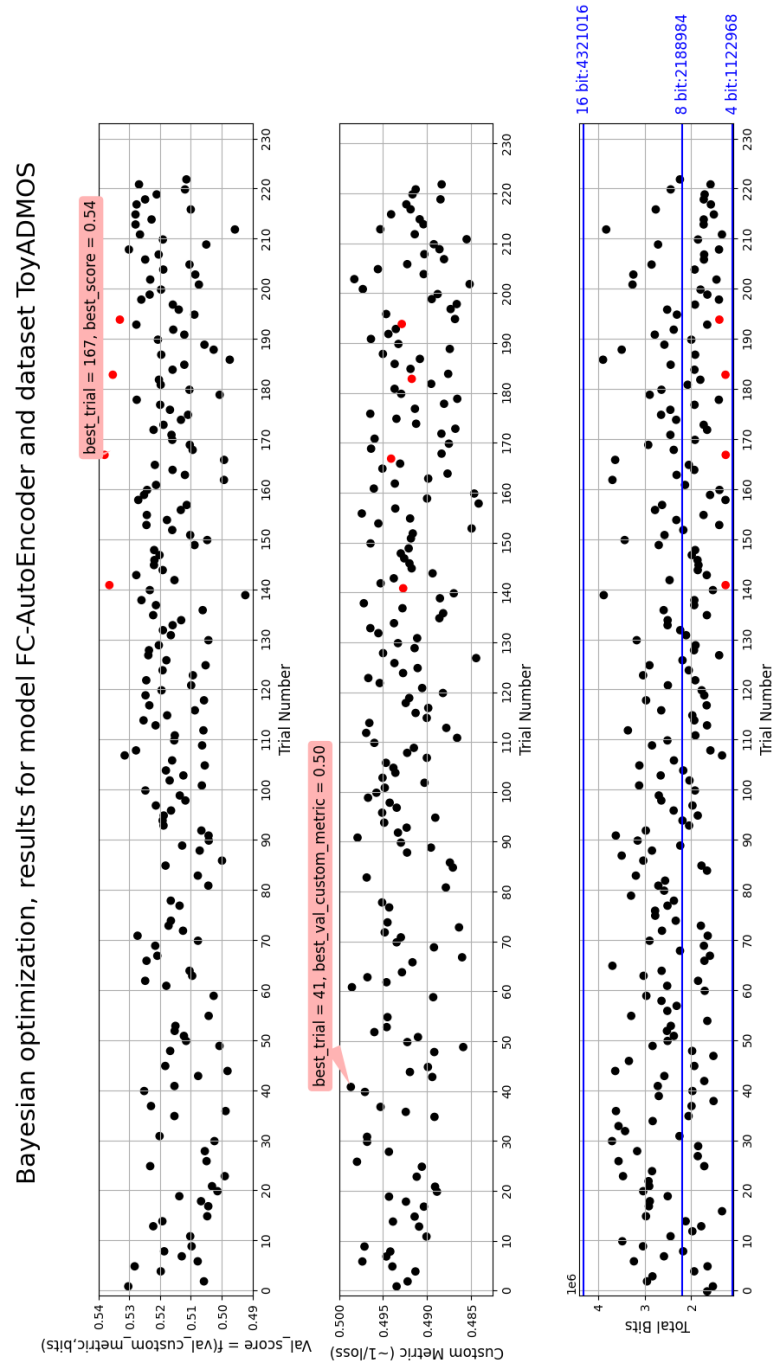


Figure 3.24: AutoQKeras search results for FC-AutoEncoder on the ToyCar ToyADMOS and MMII 248 sample perf dataset. The points within 1% of the max val score (and the corresponding points in the validation accuracy and total bits graphs) are shown in red, while the blue horizontal lines show the total reference bits of flat 16, 8 and 4 quantized models

Network	Paper test set	FP32 [%]	Our test set	FP32 Ours [%]	Accept. for sub. to contest [%]	MP [%]	16b flat [%]	8b flat [%]	4b flat [%]
MobileNet VWW	preprocessed MSCOCO 2014 test set	86.00	preprocessed MSCOCO 2014 test set	86.00	80.00	85.00	87.30	87.19	50.00
ResNet IC	CIFAR-10 perf set	86.50	CIFAR-10 perf set	88.49*	85.00	87.50	89.49	88.99	77.99
ResNet IC	N.D.	N.D.	CIFAR-10 full test set	85.92*	N.D.	84.07	84.05	84.18	76.27
DS-CNN KWS	Full Speech Commands test set	92.20	Full Speech Commands test set	92.30	90.00	90.47	91.49	91.53	73.82
AutoEncoder AD-AUC	ToyCar perf set	88.00	ToyCar perf set	88.70	88.00	87.50	86.73	88.58	76.72
AutoEncoder AD-pAUC	ToyCar perf set	N.D.	ToyCar perf set	77.28	N.D.	76.57	75.10	77.05	65.14

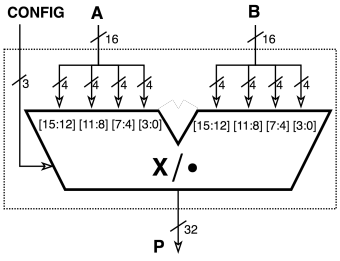
Table 3.7: Summary of MLPerf Tiny Affine Quantization results. (*We re-trained the FP model instead of importing MLPerf Tiny's weights)

Chapter 4

Reconfigurable Hardware Accelerators

In the context of embedded devices with limited processing power, tiny memory size and small energy budgets, running DNNs with relatively high accuracy is challenging. So far we have seen how mixed-precision quantization helps reduce the overall memory footprint and consequently the required power needed to run DNNs on these devices. Another current trend in modern System-on-Chips (SoCs) is to offload resource intensive DNN computations from the main processor to hardware accelerators in order to speedup the execution.

The mixed-precision networks we obtained in Ch. 3 require hardware support for variable bitwidths. In literature, several mixed-precision accelerators have been proposed for this application [41]. In this chapter we will use the ones based on Sum-Together Precision-Scalable MAC multipliers (ST PSMAC) developed with HLS techniques [38] [39]. An overview of these accelerators is shown in Fig 4.4.



(a) ST based reconfigurable multiplier

CONFIG	P
16x16 (000b)	A[15:0] x B[15:0]
16x8 (100b)	A[15:0] x B[7:0]
8x8 (010b)	A[15:8] x B[7:0] + A[7:0] x B[15:8]
8x4 (011b)	A[15:8] x B[3:0] + A[7:0] x B[11:8]
4x4 (001b)	A[15:12] x B[3:0] + A[11:8] x B[7:4] + + A[7:4] x B[11:8] + A[3:0] x B[15:12]

(b) Accuracy

Figure 4.1: Configurations for mixed-precision

This component is able to compute N = 1, 2, 4 multiplications/dot-products in parallel with operands at 16/N bits, depending on the *configuration* shown in Fig.

4.4(b). Correspondingly, the bitwidths of the input operands can be 16, 8 or 4 bits.

The main objective of this chapter is to minimize the bitwidths of the accelerator's [38] [39] internal C variables that store the DNN algorithm accumulation and affine quantization contributions, while retaining most of the test accuracy/AUC. We set 0.5% as maximum acceptable test error with respect to the original mixed-precision accuracy, across all models. Reducing these bitwidths allows the use of smaller adders and multipliers, resulting in an overall area reduction when synthesizing the three accelerators.

4.1 MLPerfTiny Cxx models

To verify if the variation of the internal C variables of the accelerators affect the test accuracy/AUC, we integrate the accelerators into Keras, defining a Python class for each accelerator, 2DConv, DWConv and FC. This class inherits the properties of the corresponding Keras layers and expands upon them by replacing the *call* method with a numpy function that in turn calls the C executable. The class attributes set the necessary parameters for the C accelerator such as input and output shapes, padding, configurations and more. Some of these require some clarification:

- config1: specifies which of the 5 computing configurations to use (Fig. 4.4 (b)) inside the accelerator
- config2: indicates which is the bitwidth of the quantized output of the accelerator, which can be 16, 8 or 4 bits
- en_relu: whether to apply the ReLU function inside the accelerator or not
- first_layer: 1 if the layer should expect dequantized (fake floating point) values at inputs
- last_layer: 1 if the layer should output a dequantized value

The new class is used to build four custom Keras models that we named *Cxx* models, where 2DConv, DWConv and FC layers are replaced by the new Cxx layers. We verified that our four Cxx models matched the equivalent QKeras models found by AutoQKeras.

Recalling that all QKeras models (with the exception of FC-AutoEncoder) have folded Batch Normalization layers, we can define the following procedure to write the equivalent Cxx model:

- for each layer set config1 and config2 using the bit configurations determined by AutoQKeras;
- whenever a 2DConv, DWConv, FC layer is followed by a ReLU activation, fold it using en_relu=1;

- if the Cxx layer is the first layer of the network, or it is preceded by a Keras layer, that works with FP numbers, enable the first_layer parameter;
- if the Cxx layer is the last one, or it is followed by a Keras layers, that works with FP numbers, enable the last_layer.

Once defined, the Cxx model can be used for inference just like a Keras model, with the only limitation being the batch size which must be 1. This is due to the fact that the C executable is currently able to process one input data at a time. In a first preliminary stage, we validated that the Cxx models gave the same results as their equivalent QKeras counterparts, making sure the predictions were the same for every test data sample. In this testing phase, the C accelerators import a header file, calleddefs_quant_max_prec.h, which sets the precision of some of the internal parameters of the accelerators to maximum precision (details in 4.2). The test data used for each model is the same as the one described in the previous chapter, except for the FC-AutoEncoder, whose inference time was deemed excessive due to the size of the dataset (248 audio wav files, where each file is made of 196 batches of 640 samples). The dataset was reduced to 84 wav files (21 randomly chosen files for each machine ID), so 4116 features per machine ID. This choice is justified by the fact that we are only interested in comparing the accuracy of the Cxx model relative to the one of QKeras.

Before running the test, to make a fair comparison between Cxx and QKeras, the latter requires that the keras backend run on float_64 precision which slightly changes the inference results from the previous chapter. Consequently, the test accuracy and AUC were re-calculated doing inference on the test sets of each QKeras model. The results are summarized in Tab. 4.1.

MobileNet	**ResNet**	**DS-CNN**	**FC-AutoEncoder**
85.00 %	87.50%	90.55%	AUC 91.75, pAUC=81.59

Table 4.1: New accuracy and AUC/pAUC values obtained doing inference on the test sets with the new tf.keras.backend

The test uses the Python *Multiprocess* library which enables parallel inference. In particular, seven processes are spawned: one for MobileNet, ResNet and DS-CNN, and four for FC-AutoEncoder, whose dataset is split among machine IDs. The test confirms the equivalence of these models.

4.2 Hardware parameter exploration

So far the accelerator's hardware parameters (which are C variables) were set to maximum precision to verify the correct implementation of the Cxx models. The

name of these C variables are reported in Tab. 4.2 which is equal to the content of defs_quant_max_prec.h, common to all accelerators. These parameters refer to the various contributions of Eq. 1.14, lumped together as shown in the table. The variables that have both total (TOT) and integer (INT) part are fixed-point variables and are defined using ac_fixed datatpye; while those that have only one value are integer variables and are defined using ac_int datatype.

Variable name in C code	**Factor**	**Max. Precision**
ACC_BITWIDTH	accumulator bitwidth	128
W_CROSS_BITWIDTH	$z_X \sum_{k=1}^{p} W_{q,k,j}$	128
SF_IN_W_TOT_BITWIDTH	$s_X s_W$	128
SF_IN_W_INT_BITWIDTH		32
BIASQ_SCALED_TOT_BITWIDTH	$s_b b_{q,j}$	128
BIASQ_SCALED_INT_BITWIDTH		32
SF_OUT_INV_TOT_BITWIDTH	$\frac{1}{s_Y}$	256
SF_OUT_INV_INT_BITWIDTH		32
Z_BITWIDTH	z_Y (z_X)	128

Table 4.2: C variable names used inside the C/C++ accelerators, their corresponding terms from Eq. 1.14, and the maximum bit precision used for each one.

Clearly, synthesizing the accelerators with maximum precision for all of these parameters is not a viable solution as the resulting hardware components would be too large (e.g. multipliers with more than 100 bits), causing high area consumption, or even unsynthesizability, because the equivalent components could not be present in standard technological libraries. This is why we need to reduce the number of bits of these variables.

Thus, we divide the space exploration of the bitwidth of the hardware parameters in two phases:

1. we determine the minimum accumulator bitwidth that ensures no error by progressively reducing it while keeping the other terms at maximum precision (Fig. 4.2).
2. we use the test sets from Sec. 4.1 as a calibration set to determine the maximum swing of each parameter per-model and per-layer. Then combine the results and determine a unique maximum and minimum value used to determine useful tests for the total and fractional bitwidths (Fig. 4.3) .

To implement this procedure, a Python script was designed to:

1. execute a TCL script that uses *awk* to replace the bitwdith of the hardware parameters in the accelerator's header file, sampling one combination of bitwidth (one column) from the search space of Figs. 4.2-4 at an iteration;

2. compile the C/C++ codes of the accelerators to apply the changes;

3. run the multiprocessing script from Sec. 4.1, collect the results and append them to a csv file.

The experiments for the accumulator bitwidth are shown in Fig. 4.2, while the experiments for the other bitwidth variations are shown in Fig. 4.3. Each column refers to a test iteration (combination of bitwidths) that was used with the Python script we just mentioned. A summary of the results obtained in the two tests is shown in Tabs. 4.3-4.4.

```
# TEST ACC_BITWIDTH AND THE REST AT HIGHEST PRECISION

# output_acc
set ACC_BITWIDTH_LIST                {40  32   28  24}
# WEIGHTS_CROSSPRODUCT
set W_CROSS_BITWIDTH_LIST            {128 128 128 128}
# SCALING_FACTOR_INPUTS_WEIGHTS
set SF_IN_W_TOT_BITWIDTH_LIST        {128 128 128 128}
set SF_IN_W_INT_BITWIDTH_LIST        { 32  32  32  32}
# BIASQ_SCALED
set BIASQ_SCALED_TOT_BITWIDTH_LIST   {128 128 128 128}
set BIASQ_SCALED_INT_BITWIDTH_LIST   { 32  32  32  32}
# SCALING_FACTOR_OUT_INVERSE
set SF_OUT_INV_TOT_BITWIDTH_LIST     {256 256 256 256}
set SF_OUT_INV_INT_BITWIDTH_LIST     { 32  32  32  32}
# Z_IN and Z_O2
set Z_BITWIDTH_LIST                  {128 128 128 128}
```

Figure 4.2: Initial accumulator bitwidth tests defined in the TCL script

```
# output_acc
set ACC_BITWIDTH_LIST                {28 28  28  28 28 28  28 28}
# WEIGHTS_CROSSPRODUCT
set W_CROSS_BITWIDTH_LIST            {35 35  35  35 35 35  35 32}
# SCALING_FACTOR_INPUTS_WEIGHTS
set SF_IN_W_TOT_BITWIDTH_LIST        {128 64 56 48  32 24  28 28}
set SF_IN_W_INT_BITWIDTH_LIST        {4    4  4  4   4  4   4  4}
# BIASQ_SCALED
set BIASQ_SCALED_TOT_BITWIDTH_LIST   {128 56 56 48  32 24  28 28}
set BIASQ_SCALED_INT_BITWIDTH_LIST   {3    3  3 3    3  3   3  3}
# SCALING_FACTOR_OUT_INVERSE
set SF_OUT_INV_TOT_BITWIDTH_LIST     {256 64 56 48  32 24  28 28}
set SF_OUT_INV_INT_BITWIDTH_LIST     {18  18 18  18 18 18  18 18}
# Z_IN and Z_O2
set Z_BITWIDTH_LIST                  {18  18 18  18 18 18  18 16}
```

Figure 4.3: Final tests defined in the TCL script

Iteration	VWW Acc	VWW Err	IC Acc	IC Err	KWS Acc	KWS Err	AD AUC	AD Err	AD pAUC	AD pAUC Err
0	85.00	0.00	87.50	0.00	90.55	0.00	91.75	0.00	81.59	0.00
1	85.00	0.00	87.50	0.00	90.55	0.00	91.75	0.00	81.59	0.00
1	85.00	0.00	87.50	0.00	90.55	0.00	91.75	0.00	81.59	0.00
3	85.00	0.00	0.05	87.45	8.34	82.21	42.10	49.65	50.39	31.20

Table 4.3: Test 0, the iterations refer to the bitwidths defined in Fig. 4.2 (columns). The error is defined with respect to the QKeras baseline accuracy/AUC

Iteration	VWW Acc	Err	IC Acc	Err	KWS Acc	Err	AD AUC	Err	AD pAUC	Err
0	85.00	0.00	87.50	0.00	90.55	0.00	91.75	0.00	81.59	0.00
1	85.00	0.00	87.50	0.00	90.55	0.00	91.75	0.00	81.59	0.00
2	85.00	0.00	87.50	0.00	90.55	0.00	91.75	0.00	81.59	0.00
3	85.00	0.00	87.50	0.00	90.55	0.00	91.75	0.00	81.59	0.00
4	85.00	0.00	87.50	0.00	90.61	-0.06	91.75	0.00	81.59	0.00
5	85.00	0.00	0.23	87.27	8.40	82.14	82.36	9.40	65.39	16.19
6	85.00	0.00	87.50	0.00	90.55	0.00	91.75	0.00	81.59	0.00
7	85.00	0.00	88.00	-0.50	90.43	0.12	40.58	51.18	50.39	31.20

Table 4.4: Test 1, the iterations refer to the bitwidths defined in Fig. 4.3 (columns). The error is defined with respect to the QKeras baseline accuracy/AUC

4.3 Accelerator speedup

In this final section we analyze how running the four networks on the ST-based reconfigurable accelerators (RECONF) could provide a latency reduction with respect to non ST-based accelerators (STANDARD), i.e. accelerators based on standard 16-bit multipliers that use sign extension whenever required for low-precision operands. Given the large data tensors involved in DNNs and the limited memory available inside these accelerators, to store input, weights and outputs, the accelerators need to divide the input tensors in chunks (called "tiles") and process them one-by-one in a sequential way. Usually in the context of a SoC, this tiling operation is performed by a processor or a Direct Memory Access (DMA) unit.

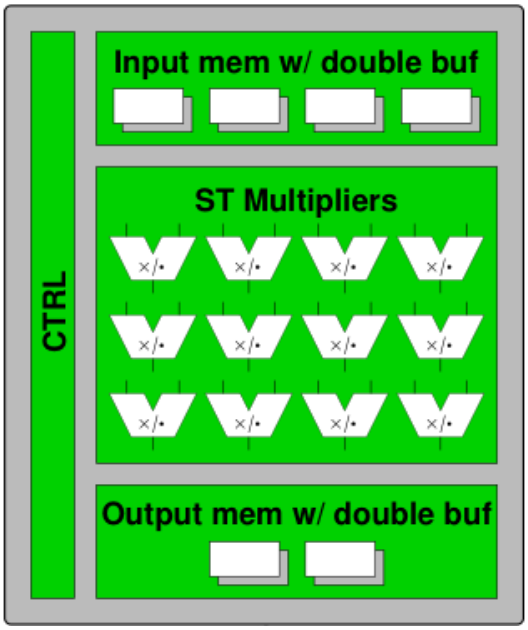


Figure 4.4: High-level overview of the accelerator in a SoC

To compute the latency we assume:

- ideal latency reduction for each reconfigurable accelerator, normalized to the one of the standard and 16-bit case:
  - STANDARD: 1 for all configurations
  - RECONF: 1 for 16x16 and 16x8, 0.5 for 8x8 and 8x4, 0.25 for 4x4
- all data transfers from external memory are transparent due to double buffering
- no overlapping between tiles

When this is true, it possible to compute the latency of a network by taking into account the total number of layers (N-1), the number of tiles required for a specific layer (T) and the configuration associated to it ($L(C_n)$) (Eq. 4.1).

$$\begin{aligned} L_{tot,RDM} &= \sum_{n=1}^{N} L_{acc,RECONF}(C_n) \times T_n \\ L_{tot,SDM} &= L_{acc,STANDARD} \sum_{n=1}^{N} T_n \end{aligned} \tag{4.1}$$

The speedup can then be computed as the ratio of the latency of the non-ST standard accelerators over the latency of the ST-based reconfigurable ones:

$$Speedup = \frac{L_{tot,STANDARD}}{L_{tot,RECONF}} \tag{4.2}$$

The speedup for the four networks is shown in Tab. 4.5. To compute the number of memory tiles required by each layer, we assume that each accelerator has the following shapes for its internal input, weight and output memories:

- 2DConv:
  - input/output memory shape = $18 \times 18 \times 16$
  - weight memory shape = $7 \times 7 \times 16 \times 16$
- DWConv:
  - input/output memory shape = $22 \times 22 \times 16$
  - weight memory shape = $5 \times 5 \times 16$
- FC:
  - input memory shape = 256
  - output memory shape = 16
  - weight memory shape = 256×16

MLPerf Tiny Network	**Latency Standard**	**Latency Reconf.**	**Speedup theoretical**
MobileNet	1155.00	505.75	2.28
ResNet	481.00	304.25	1.58
DS-CNN	2273.00	980.25	2.32
FC-AutoEncoder	121.00	44.25	2.73

Table 4.5: Speedup table

Chapter 5

Conclusion and Future Work

In this thesis we have shown how AutoQKeras can be used to find optimal mixed-precision quantized configurations for the four networks of the MLPerf Tiny Benchmark using Bayesian Optimization. These configurations provide a substantial reduction in memory footprint with respect to conventional flat 16- and 8- bit configurations, while keeping the test accuracy within a range of 2% of their floating-point counterparts. Moreover, we investigate the use of reconfigurable hardware accelerators to implement these networks and compare the results to solutions that employ standard multipliers, finding that it is possible to achieve an ideal average speedup of 2.23x with respect to standard multiplier solutions.

On a final note, we leave some hints for possible future development of this work:

- synthesize the DNN hardware accelerators [38] [39] with the optimized internal precisions found in Ch.4 using HLS techniques
- integrate those accelerators in a SoC with a RISC-V processor using ESP [30]
- run the inference of the best mixed-precision quantized MLPerf Tiny DNNs of Ch. 3 and estimate the latency reduction in different scenarios, such as either running all layers using only the processor, or only ST-based (or non-ST-based) DNN accelerators, or using a mixed approach in which some layers are mapped on the processor and others on the accelerators
- quantize in mixed-precision the same MLPerf Tiny networks with other techniques that do not use scaling factors and zero-points, such as using power-of-2 scaling or no scaling at all, to evaluate the trade-off between accuracy degradation and hardware resources utilization, using the automatic scripts developed in Ch. 2.

Bibliography

[1] Keras API.

[2] Mlperf tiny github repository. https://github.com/mlcommons/tiny.

[3] Saleh Albelwi and Ausif Mahmood. A framework for designing the architectures of deep convolutional neural networks. *Entropy*, 19(6), 2017.

[4] Saugat Bhattarai. Gradient descent. https://saugatbhattarai.com.np/what-is-gradient-descent-in-machine-learning/.

[5] Aakanksha Chowdhery, Pete Warden, Jonathon Shlens, Andrew Howard, and Rocky Rhodes. Visual wake words dataset, 2019.

[6] Claudionor N. Coelho, Aki Kuusela, Shan Li, Hao Zhuang, Thea Aarrestad, Vladimir Loncar, Jennifer Ngadiuba, Maurizio Pierini, Adrian Alan Pol, and Sioni Summers. Ultra low-latency, low-area inference accelerators using heterogeneous deep quantization with qkeras and hls4ml. 2020.

[7] Claudionor N. Coelho, Aki Kuusela, Shan Li, Hao Zhuang, Jennifer Ngadiuba, Thea Klaeboe Aarrestad, Vladimir Loncar, Maurizio Pierini, Adrian Alan Pol, and Sioni Summers. Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors. *Nature Machine Intelligence*, 3(8):675–686, jun 2021.

[8] Arden Dertat. Applied deep learning - part 3: Autoencoders. https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798.

[9] Yann LeCun et al. The mnist database of handwritten digits. http://yann.lecun.com/exdb/mnist/.

[10] Peter I. Frazier. A tutorial on bayesian optimization, 2018.

[11] Google. Autoqkeras forgiving_bits.py. https://github.com/google/qkeras/blob/3a1fb06ba40bed3b490a0942851c31405faaf007/qkeras/autoqkeras/forgiving_metrics/forgiving_bits.py#L53.

[12] Google. Autoqkeras notebook. https://github.com/google/qkeras/blob/master/notebook/AutoQKeras.ipynb.

[13] Google. Tensorboard.dev. https://tensorboard.dev/.

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[15] Matthijs Hollemans. Google's mobilenets on

the iphone. https://machinethink.net/blog/googles-mobile-net-architecture-on-iphone/.

[16] Mark Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14, 2014.

[17] 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.

[18] Martin Isaksson. Four common types of neural network layers. https://towardsdatascience.com/four-common-types-of-neural-network-layers-c0d3bb2a966c.

[19] Or Izchak. How does a neural network work? implementation and 5 examples. https://www.hotelmize.com/blog/how-does-a-neural-network-work-implementation-and-5-examples/.

[20] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference, 2017.

[21] Savya Khosla. Cnn | introduction to pooling layer. https://www.geeksforgeeks.org/cnn-introduction-to-pooling-layer/.

[22] Yuma Koizumi, Yohei Kawaguchi, Keisuke Imoto, Toshiki Nakamura, Yuki Nikaido, Ryo Tanabe, Harsh Purohit, Kaori Suefusa, Takashi Endo, Masahiro Yasuda, and Noboru Harada. Description and discussion on dcase2020 challenge task2: Unsupervised anomalous sound detection for machine condition monitoring. In *Proceedings of the Detection and Classification of Acoustic Scenes and Events 2020 Workshop (DCASE2020)*, pages 81–85, Tokyo, Japan, November 2020.

[23] Yuma Koizumi, Shoichiro Saito, Hisashi Uematsu, Noboru Harada, and Keisuke Imoto. Toyadmos: A dataset of miniature-machine operating sounds for anomalous sound detection. In *2019 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA)*, pages 313–317, 2019.

[24] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper, 2018.

[25] Alex Krizhevsky. Learning multiple layers of features from tiny images. https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

[26] Silicon Labs. Person detection. https://github.com/SiliconLabs/platform_ml_models/tree/master/eembc/Person_detection.

[27] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. 2016.

[28] David Chuan-En Lin. 8 simple techniques to prevent overfitting. https://towardsdatascience.com/

8-simple-techniques-to-prevent-overfitting-4d443da2ef7d.

[29] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. Microsoft coco: Common objects in context, 2014.

[30] Paolo Mantovani, Davide Giri, Giuseppe Di Guglielmo, Luca Piccolboni, Joseph Zuckerman, Emilio G. Cota, Michele Petracca, Christian Pilato, and Luca P. Carloni. Agile SoC development with open ESP. In *Proceedings of the 39th International Conference on Computer-Aided Design*. ACM, nov 2020.

[31] Lei Mao. Quantization for neural networks. https://leimao.github.io/article/Neural-Networks-Quantization/.

[32] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and Tijmen Blankevoort. A white paper on neural network quantization, 2021.

[33] Tom O'Malley, Elie Bursztein, James Long, François Chollet, Haifeng Jin, Luca Invernizzi, et al. Kerastuner. https://github.com/keras-team/keras-tuner, 2019.

[34] Leo Pauly, Harriet Peel, Shan Luo, David Hogg, and Raul Fuentes. Deeper networks for pavement crack detection. 07 2017.

[35] Harsh Purohit, Ryo Tanabe, Kenji Ichige, Takashi Endo, Yuki Nikaido, Kaori Suefusa, and Yohei Kawaguchi. Mimii dataset: Sound dataset for malfunctioning industrial machine investigation and inspection, 2019.

[36] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. Mlperf inference benchmark, 2019.

[37] Abhineet Saxena. Convolutional neural networks (cnns): An illustrated explanation. https://blog.xrds.acm.org/2016/06/convolutional-neural-networks-cnns-illustrated-explanation/.

[38] L. Urbinati and M. R. Casu. A reconfigurable 2d-convolution accelerator for dnns quantized with mixed-precision. In *Proceedings of International Conference on Applications in Electronics Pervading Industry, Environment and Society (ApplePies)*, Genova, Italia, 2022.

[39] L. Urbinati and M. R. Casu. A reconfigurable depth-wise convolution module for heterogeneously quantized dnns. pages 128–132, 2022.

[40] Luca Urbinati and Marco Terlizzi. Visual wake words test set github issue.

https://github.com/mlcommons/tiny/issues/135.

[41] C. Enz V. Camus, L. Mei and M. Verhelst. Review and benchmarking of precision-scalable multiply-accumulate unit architectures for embedded neural-network processing. https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

[42] Pete Warden. Speech commands: A dataset for limited-vocabulary speech recognition, 2018.

[43] Lisa Zhang. Convolutional neural networks. https://www.cs.toronto.edu/~lczhang/360/lec/w04/convnet.html.