Unsupervised Learning of Variational Autoencoders on Cortex-M Microcontrollers

Mark Deutel*†, Axel Plinge†, Dominik Seuß†‡, Christopher Mutschler†, Frank Hannig*, and Jürgen Teich*
*Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Erlangen, Germany
†Fraunhofer Institute for Integrated Circuits IIS, Nürnberg, Germany
‡Center for Artificial Intelligence and Robotics (CAIRO), Technische Hochschule Würzburg-Schweinfurt, Würzburg, Germany
{mark.deutel, frank.hannig, juergen.teich}@fau.de, {axel.plinge, dominik.seuss, christopher.mutschler}@iis.fraunhofer.de

***Abstract*—Training and fine-tuning deep neural networks (DNNs) to adapt to new and unseen data at the edge has recently attracted considerable research interest. However, most work to date has focused on supervised learning of pre-trained, quantized DNNs. Although these proposed techniques are advantageous in enabling DNN training even on resource-constrained devices such as microcontroller units (MCUs), they do not address the issue that large amounts of labeled data are required to perform supervised training. Moreover, while data are readily available at the edge, ground-truth labels are typically not. In this work, we explore variational autoencoders as a way to train unsupervised, i.e., without labels, feature extractors for image classification on a Cortex-M MCU. Using these feature extractors, we then train classification heads using small labeled representative sets of data to solve classification tasks. Our approach significantly reduces the amount of labeled data required, while enabling full DNN training from scratch on resource-constrained devices.**

***Index Terms*—Unsupervised Learning, Microcontrollers, Variational Autoencoders, Quantization-Aware Training**

I. Introduction

Edge artificial intelligence (AI) is the use of AI at the end of the cloud edge continuum [1]–[3], i.e., directly at the sensors and actuators. Here, a typical target are embedded systems with limited computing resources [4]. Thus, executing deep neural networks (DNNs) at the edge with energy-, time-, and resource-efficient inference on, e.g., microcontroller units (MCUs) has become a prevalent trend in research. Recent research has focused on finding as efficient as possible deployments of pre-trained DNNs on MCUs. However, since the environment monitored by an MCU using these deployed DNNs may be constantly changing, the DNNs must eventually be able to adapt to maintain a high quality of service even long after their initial deployment. While it is possible to retrain DNNs in the cloud and then redeploy them, this is often challenging in practice because communication between the cloud and MCUs is often unreliable and has limited throughput. This poses a problem because a significant amount of data must be transferred to the cloud to retrain the DNN, and afterward, the new set of weights must be sent back to the MCU.

As a result, recent research has explored the feasibility of training and fine-tuning quantized DNNs directly on MCUs [5], [6]. This approach overcomes the communication bottleneck because the data recorded on the MCU can be used directly to fine-tune the weights of the deployed DNN in place. However, scientific work in this context has focused only on supervised training of DNNs. The question of how to match the labels required for supervised training with the newly captured data on the MCU remains unanswered.

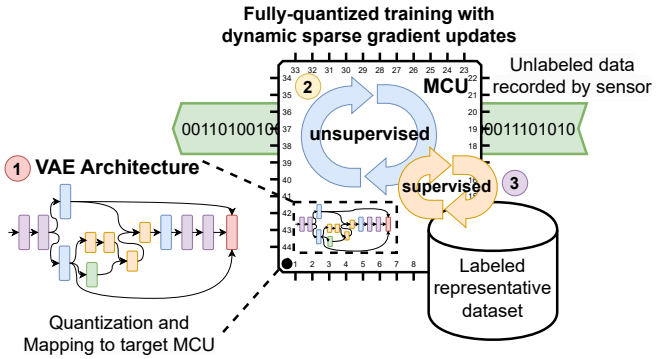


Figure 1. Proposed framework for unsupervised fully quantized training of VAEs on MCUs.

In this work, we move toward label-free, i.e., unsupervised, training of DNNs on MCUs to eliminate the need for labeled data as much as possible. To this end, we explore VAEs [7] as a means to train feature extractors for image classification tasks with only data and no labels. Thus, only a small subset of representative labeled data is needed to train a classifier in a supervised manner to perform the final classification. To the best of our knowledge, we are the first to consider such an approach for on-device training on MCUs. Our proposed framework is sketched in Fig. 1.

The contributions of our work are threefold. First, we present a methodology for deploying and training fully quantized VAEs with less than 256 kB of RAM on an MCU from scratch, that is, without any pre-training. Second, we provide a detailed analysis and discussion of the training performance of the quantized VAEs compared to their floating-point counterparts for two image classification tasks, MNIST and Fashion-MNIST. Third, we analyze the latency and energy required for VAE training on a Nucleo L4R5ZI Cortex-M4 MCU.

II. Related Work

Training DNNs on MCUs has been discussed in research mainly for supervised training to overcome the memory and computational constraints of such systems. One of the main approaches discussed is in-place training of fully quantized DNNs.

A common technique is quantization-aware training (QAT) [8]. However, QAT compensates for the additional error introduced by quantization by keeping a copy of the floating-point weights, gradients, and activations in memory, and only simulates quantization while still performing all updates to the floating-point weights. As a result, QAT cannot be feasibly applied to MCUs, where quantization is used primarily as a means to save memory, making it impossible to store floating-point copies of tensors.

Therefore, research to date has focused heavily on finding alternative approaches to performing QAT while preserving the memory savings that can be achieved with quantization. To this end, Lin *et al.* [5] propose a quantization scheme using a scaling factor called quantization-aware scaling (QAS), which they apply to the quantized gradients to compensate for the quantization error, while Deutel *et al.* [6] propose to compensate for the quantization error by normalizing the quantized gradient updates, i.e., computing the mean and standard deviation for each filter and row in convolutional and fully connected layers.

Besides quantization, another approach that has been heavily focused on in recent research to reduce the memory and computational overhead required to fine-tune a DNN is to update not all trainable weights, but only a subset. This can be done either at the "architectural" or at the "structural" level.

At the architectural level, Tiny-transfer-learning [9] freezes all weight tensors of the DNN and trains only the biases, while Frankle *et al.* [10] train only the weights of the batch normalization layers. Alternatively, the works in [5], [6], [11] focus on fine-tuning tasks by updating only the last layers of the DNN, leaving all other layers frozen at their initial pre-trained weights. Others, such as Mudrakarta *et al.* [12], perform fine-tuning by introducing small additional patches of trainable weights to a well-trained DNN.

At the structural level, Deutel *et al.* [6] update only a subset of "most important" structures, i.e., filters and rows of neurons, which the authors identify dynamically for each sample using the size of the gradient values as a heuristic. Alternatively, Lin *et al.* [5] predefine important structures within weight tensors based on gradients observed during pre-training, allowing them to perform sparse updates at runtime.

To the best of our knowledge, we are the first to consider unsupervised training of DNNs on MCUs. Nevertheless, this work builds on related work described in this section to implement a memory-efficient quantized training scheme that can be deployed on an MCU. Specifically, our work is based on the quantized training scheme proposed by Deutel *et al.* [6].

III. Method

Figure 1 shows our proposed approach for fully quantized, on-device training of VAEs on MCUs. In the first step, we quantize and map an untrained, predetermined VAE architecture to our target MCU using the deployment pipeline described in [13]. Our framework considers VAEs consisting of an encoder with convolutions and fully connected layers, and a decoder with deconvolutions and fully connected layers (see Fig. 2). In addition, we attach a small classification head to the "mean" output node μ of the VAE encoder. This classifier can then be trained

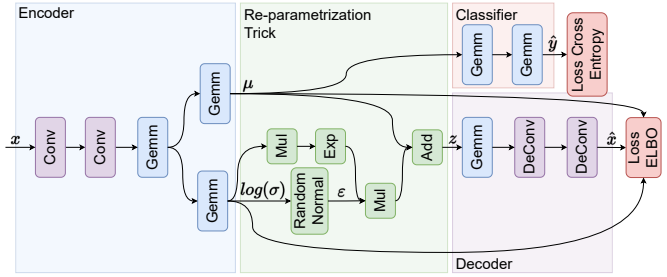


Figure 2. Architecture of the VAE evaluated in this work. The VAE consists of the encoder and the decoder with the re-parametrization trick in between. The classifier is attached to the "mean" output μ of the encoder and is trained separately.

separately from the VAE, which is trained unsupervised, on a small, separately stored support set of labeled data either stored in the MCU's flash memory or a connected external memory. In the second step, after deployment, we train the fully quantized, unsupervised VAE on the MCU using data recorded by sensor(s). For our evaluation, we simulate continuous data acquisition using a fixed training data set instead of actual sensor data. This makes our results comparable to each other and to other work. We utilize dynamic sparse gradient updates [6] to reduce the computational overhead of training. In the third step, we periodically update the small classification head attached to the encoder of our deployed VAE using the support set stored on the MCU. Once done, we return to the second step and repeat until the training has converged.

In the following, we discuss both VAE and the fully quantized training approach used in this work as part of our framework.

A. Variational Autoencoders

VAEs are a class of generative DNNs used to represent high-dimensional complex data x with a low-dimensional learned set of latent variables z. Unlike regular autoencoders (AEs), which learn a deterministic mapping, VAEs model the distribution of the latent variables z instead. However, as with autoencoders, z can be trained unsupervised, making VAEs a suitable candidate for on-device training on MCUs with only data and no labels.

Like AEs, VAEs consist of a decoder and an encoder network with a bottleneck in between. Unlike AEs, the encoder in a VAE learns a mapping of x to a Gaussian distribution $q(z|x)$, i.e., the encoder outputs vectors of means and standard deviations. The decoder, on the other hand, generates new data $\hat{x}$ based on z such that the newly generated data resembles the original data x as close as possible.

VAEs can be trained using the evidence lower bound (ELBO) loss, see Eq. (1), which is a lower bound on the log-likelihood of x, where $p(x|z)$ is the conditional distribution of z given x and KL is the Kullback-Leibler divergence between q and p:

$$\text{ELBO}(q) = \mathbb{E}_q[\log p(x|z)] - \text{KL}(q(z)||p(z)) \tag{1}$$

Since Gaussian distributions, like q, are generally not differentiable as discrete functions, the reparameterization trick is used to allow the application of backpropagation, see Eq. (2):

$$z = \mu + \sigma\varepsilon, \;\; \varepsilon \sim \mathcal{N}(0,1) \tag{2}$$

The reparameterization trick is based on the observation that all Gaussian distributions are scaled and translated versions of the normal distribution $\mathcal{N}(0,1)$. In Eq. (2), this allows the stochasticity in z to be isolated to ε. As a result, backpropagation through ε is not required, and the VAE can be trained using regular gradient descent.

In summary, in the context of VAEs, the first part of the ELBO loss can be considered as the reconstruction term between the original data x given to the encoder and the data $\hat{x}$ reconstructed by the decoder, while the Kullback-Leibler divergence term regularises z to be more stochastic.

To use a VAE for classification, we add a classifier, i.e., two additional fully connected layers, that map from the "mean" output node μ of the encoder network to a class probability vector $\hat{y}$. In this way, the encoder can be seen as a feature extractor that can be trained unsupervised using the decoder. To train the classifier, we then only need a small representative set of labeled data, since the feature extractor has already learned a mapping to a compact representative latent space beforehand.

B. Fully-Quantized Training

To reduce the memory requirements for on-device training of VAEs and to make their use on MCUs feasible, we train both the encoder and decoder fully quantized. This means that all trainable weights, as well as all activation and gradient tensors, are represented in memory as 8-bit unsigned integers v_q, which are sampled from the original floating-point values v_f using a linear quantization scheme $v_q = (v_f/s) + zp$ with adjustable zero-point z and scale s parameters. The only part of the VAE that remains in floating-point is the re-parametrization trick between the encoder and the decoder. This allows us to maximize memory savings while still being able to calculate the ELBO loss accurately.

During testing, we observed a high variance in the distribution of values between different activation, gradient, weight, and bias tensors. As a result, and as is common for quantized DNNs, we implement different zero-point and scale parameters for each tensor. This allows our approach to make optimal use of the limited 8-bit integer range to support quantized tensors with different value distributions.

Since the optimal scale and zero point of a tensor cannot be known in advance because it depends on the training data, the parameters must be inferred online during training. For example, in QATs [8], the scale and zero point parameters for a given tensor are computed based on the maximum $f_{\max}$ and minimum $f_{\min}$ floating-point values in the tensor. Since the ranges of the stored values change over the course of training as the weights of the DNN are adjusted, and also because they depend on other aspects such as the distribution of values in the received input, the $f_{\max}$ and $f_{\min}$ values are often tracked over time using a moving average whose update rate α is a hyperparameter. Based on the tracked $f_{\max}$ and $f_{\min}$ values, the corresponding scale and zero point parameters at state i are recalculated periodically, i.e. to state $i+1$, after each batch using Eqs. (3) and (4).

$$s_{i+1} = \frac{f_{\max} - f_{\min}}{255} \tag{3}$$

$$z_{i+1} = \left\lfloor -\frac{f_{\min}}{s_{i+1}} \right\rceil \tag{4}$$

An important difference between our approach and other QATs implementations, where the quantization is simulated with the floating-point tensors still existing as copies, is that in our case, all weights, activations, and gradient tensors are only stored quantized in memory. As a result, our approach requires a different strategy to keep track of the exact $f_{\max}$ and $f_{\min}$ values per tensor without the quantization error of the current scale and zero point parameters already added.

To better describe how our method keeps track of the $f_{\max}$ and $f_{\min}$ values, we discuss as an example how our method is used to compute and update the zero point and scale parameters of the error tensor $E_{n-1} = W_n^{\mathrm{T}} \cdot E_n$ of a fully connected or convolutional layer at index n, where the $\cdot$ operator is either a matrix multiplication or a 2D convolution, and W_n are the trainable weights of the layer.

When calculating E_{n-1} fully quantized, W_n and E_n must first be dequantized using their respective scale parameters s_{w_n} and s_{e_n} and zero point parameters z_{w_n} and z_{e_n} before the matrix multiplication or 2D convolution can be performed in floating-point. The result can then be quantized again to calculate the output E_{n-1} using $s_{e_{n-1}}$ and $z_{e_{n-1}}$. This process can be optimized to be computed almost entirely in integer, see Eqs. (5) and (6), with only the temporary variable e^f_{n-1} stored as floating-point numbers.

$$e^f_{n-1} = s_{w_n} s_{e_n} \sum (w_n - z_{w_n}) \cdot (e_n - z_{e_n}) \tag{5}$$

$$e_{n-1} = \left\lfloor \frac{1}{s_{e_{n-1}}} e^f_{n-1} \right\rceil + z_{e_{n-1}} \tag{6}$$

Since E_{n-1} is computed value by value, $f_{\max}$ and $f_{\min}$ can be updated iteratively by comparing whether the current value e^f_{n-1} is either greater or smaller than any of the other e^f_{n-1} values seen up to that point. As a result, once all elements of E_{n-1} have been computed, $f_{\max}$ and $f_{\min}$ accurately store the minimum and maximum values of E_{n-1} in floating-point space without the quantization error introduced by $s_{e_{n-1}}$ and $z_{e_{n-1}}$, while at the same time not requiring the storage of complete tensors of intermediate e^f_{n-1} floating-point values in RAM.

C. Dynamic Sparse Gradient Updates

To make quantized VAE training computationally efficient, we implement dynamic sparse gradient updates (DSGU), a technique first proposed in [6]. The technique is based on the observation that for a given sample, not all neurons in the layers of a DNN receive a significant update, but only a subset of neurons that are strongly activated by the sample during the forward pass. Furthermore, for DNNs, the error tensors generally become sparser after each layer during backpropagation. As a result, the approach estimates the importance of updating clusters of neurons, i.e., rows in fully connected layers and filters in

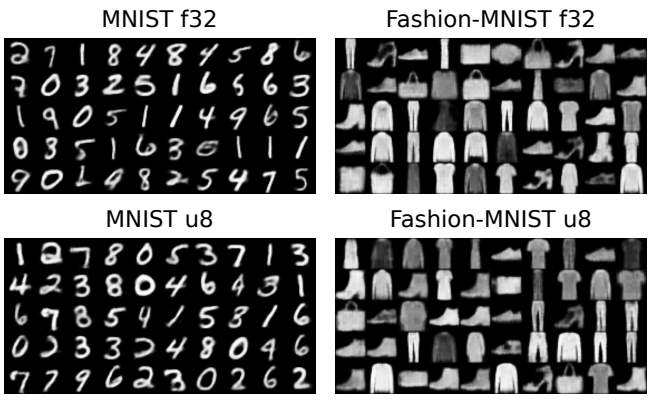


Figure 3. Examples of reproduced images generated by the baseline VAE decoder for MNIST and Fashion MNIST, after 100 epochs of floating-point (f32) and quantized (u8) training.

convolutional layers, using the absolute magnitude of the values in the error tensors of each layer as a heuristic, see Eq. (7):

$$k = \max\left\{ \left\lfloor \left(\lambda_{\min} + \left| \frac{e}{e_{\max}} \right| (\lambda_{\max} - \lambda_{\min}) \right) N \right\rfloor, 1 \right\} \quad (7)$$

The equation computes the number of structures k to be updated during backpropagation of a given layer with N total structures between given upper and lower limits $0 < \lambda_{\min} < \lambda_{\max} \leq 1$ based on the current average error e of the layer and the maximum average error $e_{\max}$ observed for the layer over the entire training.

IV. Evaluation

We trained our VAE on two datasets, MNIST [14] and Fashion-MNIST [15], on a Cortex-M4 MCU using a learning rate of 0.001 and a batch size of 100. Both datasets have 10 classes each and consist of small grayscale images ($1 \times 28 \times 28$). MNIST has handwritten digits, and Fashion-MNIST has different types of clothing. Both datasets have 60 000 samples for training and 10 000 separate samples for validation. For training the VAE, we used the full training set without the labels, while for training the classification head we selected a balanced subset including the labels. For validation of both the VAE and the classification head, we used the complete labeled test set.

To evaluate their performance, we deployed the VAEs both in floating-point and quantized form. The objective of the evaluation is to (a) discuss the impact different architectural trade-offs have on both the training performance of the VAEs as well as their resource requirements, (b) evaluate if fully quantized training of VAEs can match the training accuracy of floating-point training, and (c) discuss if the saving of resources achieved by quantized training leads to feasible trade-offs that improve the deployment of VAEs and enable unsupervised training on MCUs.

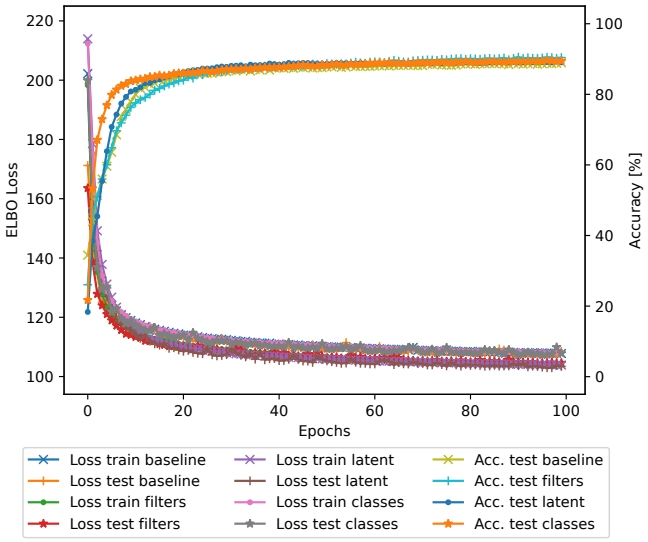
A. Floating-point On-Device Training

We evaluate four variants of VAE architectures and discuss the impact of changes in these variants on the ELBO loss of the VAE and the accuracy of the classifier. The **baseline** VAE architecture, variant 1, consists of two convolutional layers followed by a fully connected layer as the encoder, and a decoder with a fully connected layer followed by two deconvolutions, see Fig. 2. Furthermore, the latent space between the encoder and decoder has 10 dimensions, and the classifier consists of two fully connected layers with a total of 400 neurons. In variant 2 (**filters**) we double the number of filters in the convolutions and deconvolutions, in variant 3 (**latent**) we increase the size of the latent space to 40, and in variant 4 (**classes**) we increase the number of neurons in the classifier to 1920. We use these four architectures throughout the evaluation and refer to them by the boldface names given here.

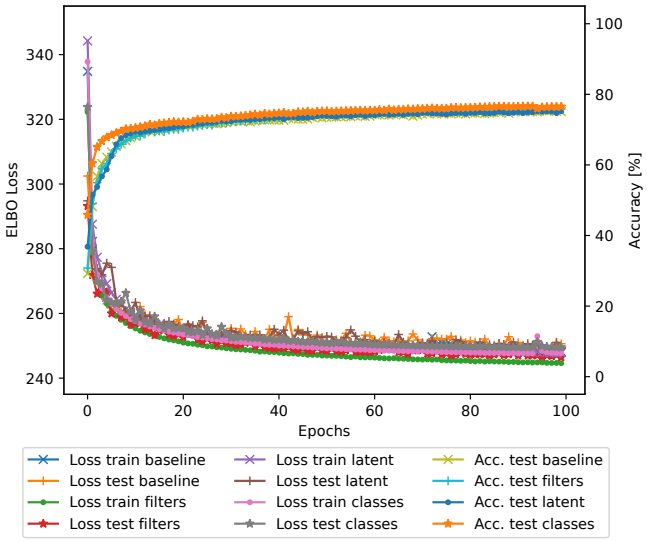
In Figs. 4a and 4b, we show the ELBO loss and the test accuracy of the classifier over 100 epochs of training with the VAEs deployed in floating-point. Furthermore, in Fig. 3, we show examples of images reproduced by the decoder after training as additional qualitative results. After each epoch of unlabeled VAE training, see the ELBO loss in the figures, we froze the encoder and decoder and then first fine-tuned the classifier using a balanced set of labeled data for 25 epochs and then evaluated it using the full test datasets. Additionally, in Figs. 4c and 4d we show the training of the classifier of the baseline VAE variant with three numbers of labeled training samples per class for both MNIST and Fashion-MNIST.

In all our results, the ELBO loss achieved by all four architectural variants converged to a similar lower bound. Furthermore, the accuracy achieved by the classifiers eventually converged to a similar maximum accuracy given enough training epochs. However, we noticed in our experiments that especially increasing the number of neurons in the classifier, see orange accuracy curves, accelerated the increase in accuracy during the first 20 training epochs, while changes in the VAE architecture, i.e., the other variants, had a negligible effect on the classifier's accuracy. Overall, the results we achieved with unsupervised training is not bad, but it is about 10% lower than what could be achieved with a CNN trained on the two datasets fully supervised. We conclude that either the architectural changes we tested in our results would need to be much larger to have a significant effect, which would then make deployment on MCUs infeasible, or that the training technique itself, i.e., using VAEs that learn a latent space unsupervised and combining them with a classifier trained supervised, is limited and cannot achieve the same results as supervised training.

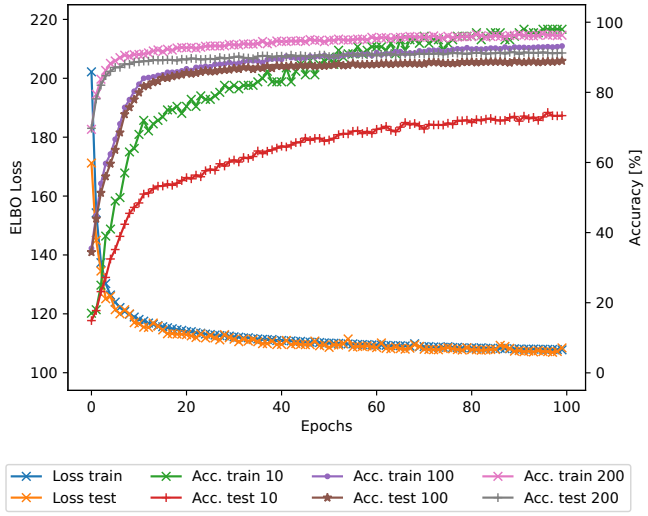
Additionally, we trained the classifier of the baseline VAE variant with labeled datasets of different sizes, 10, 100, and 200 samples per class, see Fig. 4c and 4d. In the two figures, we report both the training accuracy achieved using these small labeled datasets and the test accuracy achieved using the full MNIST and Fashion-MNIST test datasets. For both datasets, training with 10 samples per class, while yielding the highest accuracy on the training datasets, resulted in significantly lower accuracy on the full test datasets. This is a clear indication that while the model learned the small set of labeled samples from the training dataset well, it did not learn to abstract, i.e., the classifier overfitted on the training data, ultimately leading



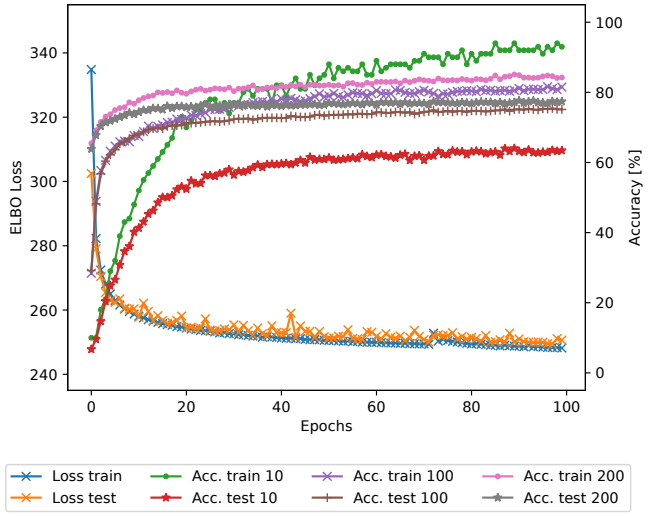
(a) Architectural variants trained on MNIST



(b) Architectural variants trained on Fashion-MNIST



(c) Different number of labeled samples trained on MNIST



(d) Different number of labeled samples trained on Fashion-MNIST

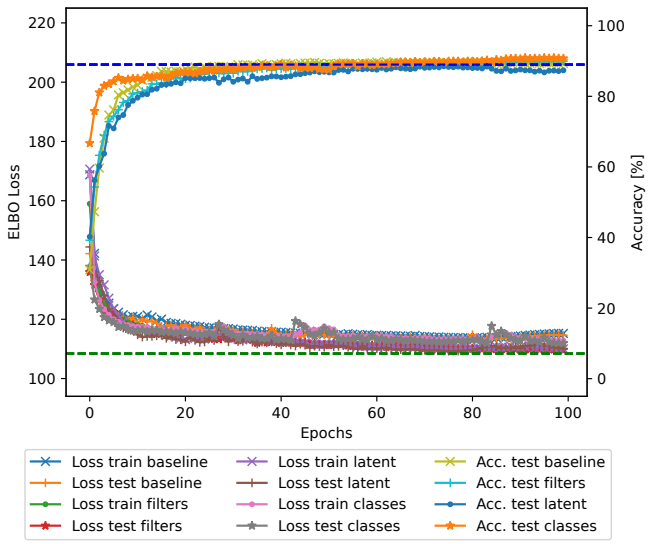
Figure 4. VAEs trained in floating-point on MNIST and Fashion-MNIST. In (a) and (b), we show the ELBO loss and accuracy of four different architectural variants, while in (c) and (d), we show the impact different numbers of labeled samples, i.e., 10, 100, and 200 samples per class, have on the accuracy achieved by the classifier.

to much worse performance on the unseen test data. For 100 and 200 samples per class, the difference between training and test accuracy is much smaller, as the classifier performed much better on the test data. When using 200 samples per class, the test accuracy is slightly higher than when using only 100 samples. However, since the difference is relatively small for both datasets, i.e., less than 2% for both MNIST and Fashion-MNIST, we still argue that having about 100 labeled samples per class is a good compromise between having the classifier learn an abstract mapping from latent space to a class probability vector and minimizing the number of labeled samples used for supervised training.

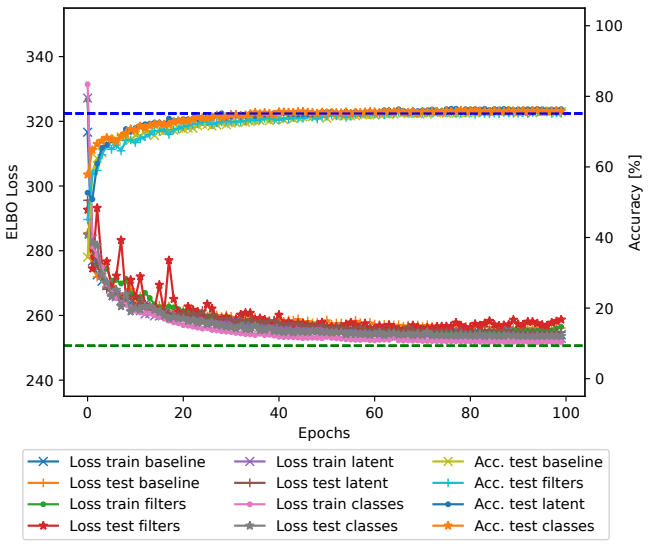
B. *Fully Quantized On-Device Training*

For quantized training, we evaluated the same four VAE variants as for floating-point, but with all trainable weights as well as activations and gradients fully quantized, except for the reparametrization trick and the ELBO loss, which we still compute in floating-point, see Fig. 2 for reference. We also kept the classifier weights in float. We did this because the number of trainable parameters in the classifier, i.e., 400 trainable weights, is so small compared to the number of trainable parameters in the VAE that it does not significantly affect the overall memory requirements or computational overhead.

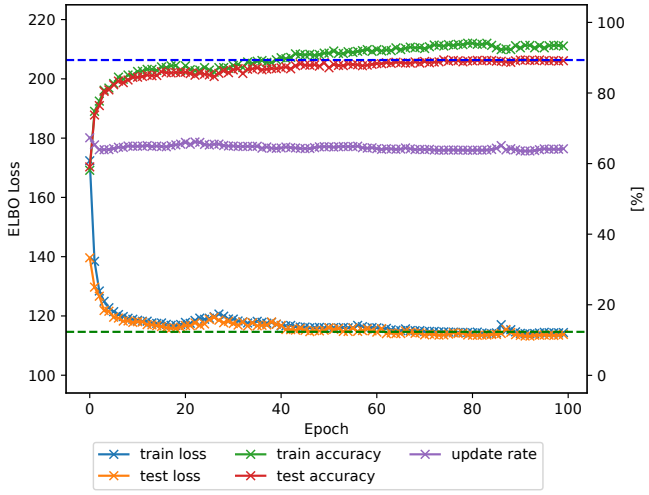
Our quantization algorithm introduces three additional hyperparameters into the training, α_e (error), α_a (activations), α_w



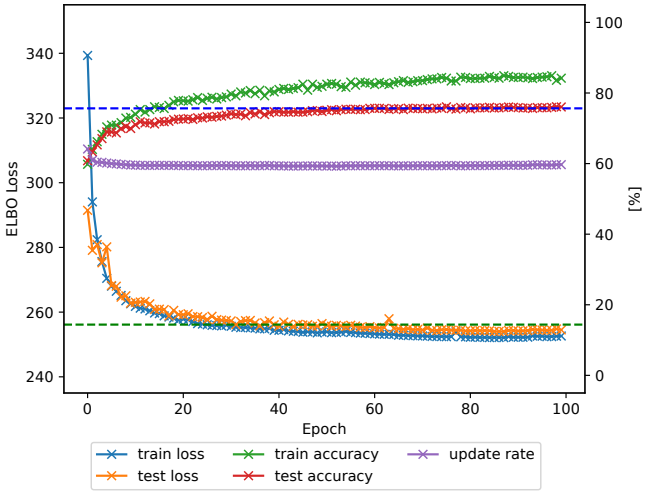
(a) VAE variants trained on MNIST



(b) VAE variants trained on Fashion-MNIST



(c) Baseline VAE variant trained on MNIST with DSGU



(d) Baseline VAE variant trained on Fashion-MNIST with DSGU

Figure 5. Quantized VAEs variants trained on MNIST and Fashion-MNIST. In (a) and (b), we show the ELBO loss and accuracy of the four different architectural variants we tested. The green and blue dashed lines denote the best loss and accuracy achieved by the baseline VAE variant when trained in floating-point, see Fig. 4. In (c) and (d), we show results for training the baseline VAE variant with DSGU enabled. The dashed lines indicate the best loss and accuracy of the quantized baseline VAE variant from (a) and (b).

(weights), and α_{norm} (weight normalization), which control the moving averages for both the $f_{\min}$ and $f_{\max}$ parameters, and the mean and standard deviation parameters used for weight normalization, as described in [6]. In our experiments, setting $\alpha_e = \alpha_a = 0.1$, $\alpha_w = 0.001$, and $\alpha_{norm} = 0.85$ for MNIST and $\alpha_{norm} = 0.7$ for FMNIST gave the best results.

Since we already established in the last section that 100 labeled samples per class is a good compromise to achieve the highest accuracy with the lowest number of labeled samples, we performed all experiments with quantized VAE using labeled datasets with 100 samples per class.

As with the floating-point results presented in the previous section, we show the ELBO loss of the quantized VAE as well as the test accuracy of the classifier for MNIST and Fashion-MNIST in Figs. 5a and 5b. In the figures, we have also marked the final loss and accuracy achieved by the floating-point VAE baseline variant from Figs. 4a and 4b with green and blue dashed vertical lines as a reference. For the four VAE variants tested, the results show that by introducing quantization, all VAE variants generally still converged to a similar lower bound as with floating-point during training. The direct comparison between the quantized and floating-point baseline VAE variant shows only a slight degradation in ELBO loss for both MNIST and Fashion-MNIST after 100 training epochs (compare the blue and orange loss curves in Figs. 5a and 5b with the green dashed vertical lines). However, despite the slightly higher loss of the

quantized VAEs, the accuracy achieved by their classifiers is still the same for both datasets, i.e., $< 0.2\%$ difference in accuracy on the test dataset; compare the accuracy curves with the blue dashed lines. In addition, comparing the qualitative results of the decoder in Fig. 3, no significant degradation in the quality of the reproductions can be seen between floating-point and quantized training.

We also evaluated training the quantized VAE variants in combination with dynamic sparse gradient updating (DSGU), see Figs. 5c and 5d. In addition to the ELBO loss and accuracy, we also show the average update rate per epoch, represented by the purple curves. The update rate is dynamically calculated for each layer based on the absolute magnitude of the error tensor values received by the layer during backpropagation. It determines, on a per-sample basis, the percentage of structures, i.e., filters and rows, for which an update is computed during backpropagation. As a result, it can significantly reduce the latency of training, e.g., an average update rate of 60% results in a $1.2\times$ speedup, while an average update rate of 10% results in a $2.0\times$ speedup; compare the measured latency values for u8 baseline 60% and 10% with those of u8 baseline in Fig. 6. Furthermore, in Figs. 5c and 5d we have marked the final loss and accuracy achieved by quantized training of the baseline VAE variant when performing full gradient updates with green and blue dashed lines for reference.

The results show that VAE training converged to the same loss and accuracy for both MNIST and Fashion-MNIST with only a fraction of the gradient updates actually computed (see the purple curves). For both datasets, after an initial slightly higher update rate during the first 5 epochs, the update rates quickly plateaued at around 60% for Fashion-MNIST and slightly higher at around 63% for MNIST. In this respect, the update rate follows the loss curves relatively closely, comparing the purple with the orange and blue curves, which is to be expected since the update rate we report is derived from the error rate per sample, which in turn is calculated from the loss.

C. Comparing Memory, Latency, and Energy

We compare the memory requirements, latency, and energy of the quantized VAEs to their floating-point counterparts, see Fig. 6 and Table I. We only report SRAM requirements for memory because the VAEs are trainable, and therefore, their weights are all placed in SRAM. We distinguish two memory segments: *activations*, which contains the memory required for activations and error tensors, and *weights*, which contains all trainable weights and their gradient buffers. For energy and latency, we report results measured on a Nucleo L4R5ZI Cortex-M4 MCU. The MCU has 640 kB SRAM and a clock speed of 120 MHz. To measure the power consumption of the MCU, we used a Joulescope energy analyzer, which allowed us to sample both current and voltage at the JP5 (IDD) jumper of the development board at 1 MHz. We also recorded the latency of each training step as well as only the inference of each step externally using two GPIO pins connected to the joulescope.

For the four VAE architectures, we achieved memory savings between 49.5% and 57.5% when using quantization compared to using floating-point to store weights and activations. As a result, the size of the quantized models is reduced so much that they all fit into the memory of the Nucleo L4R5ZI, compared to the floating-point VAEs, where only the baseline (and classes) variant requires less than 640 kB of SRAM, see Tab. I. Therefore, we only measured the baseline floating-point model variant on the target MCU, while for the quantized models we were able to execute and measure all variants on device.

In addition to memory, we also analyze latency and energy during training on the MCU for both the quantized and floating-point VAEs. In Fig. 6, we show the average latency, performance, and energy for a single training step, i.e., forward and backward pass. First, it can be seen that for all evaluated models, inference takes a significantly smaller part of the total time to perform a training step compared to the backward pass. This is because backpropagation requires about twice as many operations as inference since two partial gradients with respect to error and weights have to be computed for all trainable layers. Second,

Table I
MEMORY REQUIRED FOR TRAINING THE QUANTIZED AND FLOATING-POINT VAES. ACTIVATIONS AND WEIGHTS ARE BOTH STORED IN RAM.

Model	Type	Activations [kB]	Weights [kB]
baseline	float32	45.3	484.1
	uint8	23.0 (↓ 49.2%)	221.0 (↓ 54.3%)
latent	float32	46.4	742.6
	uint8	23.6 (↓ 49.1%)	347.4 (↓ 53.2%)
filters	float32	84.3	1201.4
	uint8	36.0 (↓ 57.3%)	590.3 (↓ 50.9%)
classes	float32	46.4	496.2
	uint8	24.0 (↓ 48.3%)	236.2 (↓ 52.4%)

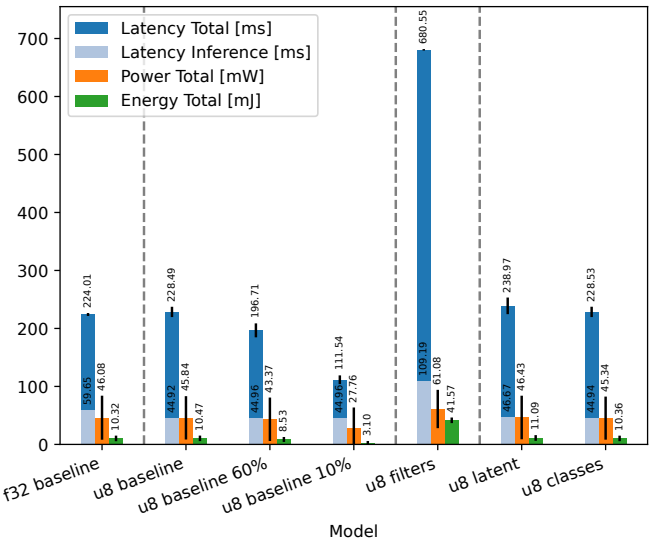


Figure 6. Average latency, power, and energy measured for floating-point (f32) and quantized (u8) VAEs variants on a Nucleo L4R5ZI Cortex-M4 MCU. "Total" indicates that the measurement was taken over the course of a full training step, while "Inference" indicates that it was taken only during the inference of a sample. We also provide results for the quantized baseline architecture when using DSGU and update rates of 60% and 10%, respectively.

we note that, unlike inference, where applying quantization resulted in a speedup (compare the light blue bars between the f32 baseline and the u8 baseline in Fig. 6) for the complete training step the overall latency is similar between the floating-point and quantized VAEs.

This is a consequence of the way quantized training is implemented in our proposed method; in particular, the dynamic adaptation of the zero-point and scale parameters is necessary only for quantized training but not for floating-point training. To compute the f_{min} and f_{max} values for each tensor, a significant overhead of additional compare operations is introduced. Furthermore, our implementation uses SIMD instructions from ARM's digital signal processor (DSP) extension during the forward pass, but not during the backward pass. This is because the memory layout of weights and activations is highly optimized to achieve maximum speedup during inference, but not during training, where transposed versions of the respective operations are computed, especially for convolutional and fully connected layers. This makes it much more difficult to use SIMD operations without either copying data or changing the memory layout. As a result, our evaluation shows that the speedup achieved by quantized inference is consumed by the additional overhead required for quantized training.

The same observations that we described for the quantized baseline VAE variant in Fig. 6 can also be observed for the other quantized VAE variants, i.e., classes, filters, and latent. We want to particularly explain the results of the filters VAE variant, which show that especially increasing the number of filters in the convolutions and deconvolutions has a significant impact on the latency since these operations are computationally expensive compared to increasing the dimensions of the latent space or the number of neurons in the classifier, which have a much smaller impact on the latency.

To summarize our results, our proposed training approach for quantized VAEs can achieve the same accuracy and ELBO loss as floating-point training, while achieving significant memory savings of up to 57%. In our experiments, while quantization did speed up inference, it did not provide a significant speedup (or energy savings) for training due to the overhead of updating the zero-point and scaling parameters. Instead, we used DSGU to speed up training, on average by a factor of $1.2\times$ in our experiments. In conclusion, we argue that for many applications, quantized training of VAEs on MCUs is advantageous. We argue that during the deployment of a VAE, it is likely that the model will be used most of the time to perform inference on samples, and only sporadically trained in between. As a result, the lower latency of quantized inference, DSGU, and the significant memory savings that can be achieved by our proposed quantized training approach offer a better tradeoff for on-device VAE training compared to floating-point training.

V. Conclusion

In this work, we proposed a method for unsupervised training of fully quantized DNNs on MCUs using VAEs. We evaluated our method on two datasets, MNIST and Fashion-MNIST, and discussed different architectural variants as well as the impact of quantization on accuracy, memory, latency, and energy. Our results show that by introducing quantization to integer values, VAEs can still be trained with the same accuracy as floating-point values, while requiring significantly less memory. This makes deploying and training VAEs on a wide range of different MCUs feasible.

References

[1] R. Singh and S. S. Gill, "Edge AI: A survey," *Internet of Things and Cyber-Physical Systems*, vol. 3, pp. 71–92, 2023. DOI: 10.1016/j.iotcps.2023.02.004.

[2] S. S. Gill, M. Golec, J. Hu, M. Xu, J. Du, H. Wu, G. K. Walia, S. S. Murugesan, B. Ali, M. Kumar, K. Ye, P. Verma, S. Kumar, F. Cuadrado, and S. Uhlig, "Edge AI: A taxonomy, systematic review and future directions," *Cluster Computing*, vol. 28, no. 18, 2025. DOI: 10.1007/s10586-024-04686-y.

[3] N. Witt, M. Deutel, J. Schubert, C. Sobel, and P. Woller, "Energy-efficient AI on the edge," in *Unlocking Artificial Intelligence: From Theory to Applications*, Springer, 2024, pp. 359–380. DOI: 10.1007/978-3-031-64832-8_19.

[4] K. Morik, J. Rahnenführer, and C. Wietfeld, Eds., *Machine Learning under Resource Constraints - Applications*. Berlin, Boston: De Gruyter, 2023, vol. 3. DOI: doi:10.1515/9783110785982.

[5] J. Lin, L. Zhu, W.-M. Chen, W.-C. Wang, C. Gan, and S. Han, "On-device training under 256kb memory," *Advances in Neural Information Processing Systems*, vol. 35, pp. 22 941–22 954, 2022.

[6] M. Deutel, F. Hannig, C. Mutschler, and J. Teich, "On-device training of fully quantized deep neural networks on Cortex-M microcontrollers," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 44, no. 4, pp. 1250–1261, 2025. DOI: 10.1109/TCAD.2024.3484354.

[7] D. P. Kingma and M. Welling, "Auto-encoding variational Bayes," *The Computing Research Repository (CoRR)*, 2022. arXiv: 1312.6114v11 [stat.ML].

[8] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2018, pp. 2704–2713. DOI: 10.1109/CVPR.2018.00286.

[9] H. Cai, C. Gan, L. Zhu, and S. Han, "TinyTL: Reduce memory, not parameters for efficient on-device learning," in *Advances in Neural Information Processing Systems*, vol. 33, Curran Associates, Inc., 2020, pp. 11 285–11 297.

[10] J. Frankle, D. J. Schwab, and A. S. Morcos, "Training batchnorm and only batchnorm: On the expressive power of random features in CNNs," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021.

[11] H. Ren, D. Anicic, and T. A. Runkler, "TinyOL: TinyML with online-learning on microcontrollers," in *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, IEEE, 2021, pp. 1–8. DOI: 10.1109/IJCNN52387.2021.9533927.

[12] P. K. Mudrakarta, M. Sandler, A. Zhmoginov, and A. Howard, "K for the price of 1: Parameter-efficient multi-task and transfer learning," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2019.

[13] M. Deutel, P. Woller, C. Mutschler, and J. Teich, "Energy-efficient deployment of deep learning applications on cortex-m based microcontrollers using deep compression," in *In Proceedings of the 26th Workshop on Methods and Description Languages for Modelling and Verification of Circuits and Systems (MBMV)*, 2023, pp. 13–24.

[14] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. DOI: 10.1109/5.726791.

[15] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: A novel image dataset for benchmarking machine learning algorithms," *The Computing Research Repository (CoRR)*, 2017. arXiv: 1708.07747 [cs.LG].