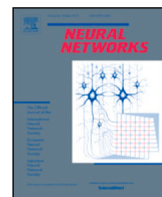
Neural Networks 191 (2025) 107763

Contents lists available at ScienceDirect

Neural Networks

journal homepage: www.elsevier.com/locate/neunet



Full Length Article

Neural networks with low-resolution parameters

Eduardo Lobo Lustosa Cabral [a], Larissa Driemeier [b,*]

[a] Institute for Energy and Nuclear Research and Mauá Institute of Technology, São Paulo, SP, Brazil

[b] Department of Mechatronics and Mechanical Systems Engineering - Polytechnic School - University of São Paulo, São Paulo, SP, Brazil

ARTICLE INFO

Keywords:
Deep learning
Low resolution
Weight quantization

ABSTRACT

The expanding scale of large neural network models introduces significant challenges, driving efforts to reduce memory usage and enhance computational efficiency. Such measures are crucial to ensure the practical implementation and effective application of these sophisticated models across a wide array of use cases. This study examines the impact of parameter bit precision on model performance compared to standard 32-bit models, with a focus on multiclass object classification in images. The models analyzed include those with fully connected layers, convolutional layers, and transformer blocks, with model weight resolution ranging from 1 bit to 4.08 bits. The findings indicate that models with lower parameter bit precision achieve results comparable to 32-bit models, showing promise for use in memory-constrained devices. While low-resolution models with a small number of parameters require more training epochs to achieve accuracy comparable to 32-bit models, those with a large number of parameters achieve similar performance within the same number of epochs. Additionally, data augmentation can destabilize training in low-resolution models, but including zero as a potential value in the weight parameters helps maintain stability and prevents performance degradation. Overall, 2.32-bit weights offer the optimal balance of memory reduction, performance, and efficiency. However, further research should explore other dataset types and more complex and larger models. These findings suggest a potential new era for optimized neural network models with reduced memory requirements and improved computational efficiency, though advancements in dedicated hardware are necessary to fully realize this potential.

1. Introduction

Deep neural networks (DNNs) are fundamental to numerous recent advancements in Artificial Intelligence (AI), playing a central role in the emergence of foundation models and generative AI. One of the largest neural networks currently in use is the Megatron-Turing NGL 530B, a generative language model developed by Nvidia and Microsoft, boasting 530 billion parameters. However, deploying advanced machine learning models poses numerous complex engineering challenges due to the need for powerful computational devices and large memory storage. This not only affects training but also renders it unfeasible to run on devices with limited computational resources (Dai et al., 2023).

The pioneering works by Gupta et al. (2015) and Courbariaux et al. (2015) originated from the idea that leveraging the inherent noise-tolerance of neural network algorithms could allow for the relaxation of certain constraints on underlying hardware. Gupta et al. (2015) specifically explored the use of low-precision fixed-point arithmetic for training deep neural networks, with a particular focus on the rounding mode used during operations on fixed-point numbers. Courbariaux et al. (2015) analyzed the performance of a Maxout network in three benchmark datasets with three distinct formats: floating point, fixed point and dynamic fixed point. The authors compared the performance with results from literature.

Lower precision means each number uses fewer bits, which makes memory smaller, cheaper, and more power-efficient. Fixed-point arithmetic is simpler and faster than floating-point, especially with low precision. This reduces the load on processors, allowing them to perform more operations per second and improving overall efficiency. Using less power extends battery life in portable devices and lowers cooling needs, reducing system costs and complexity. Lower precision can lead to more affordable and accessible hardware, enabling wider use of neural network applications.

Inspired on the works by Gupta et al. (2015) and Courbariaux et al. (2015), it is natural to quantize parameters to optimize memory usage and improve computational efficiency, instead of reducing their total number in large AI models. Quantization decreases the bit width of the values used, thereby reducing the computational cost of floating-point operations. For example, in Moosmann et al. (2024), the authors

* Corresponding author.
E-mail addresses: elcabral@ipen.br, elcabral@maua.br (E.L.L. Cabral), driemeie@usp.br (L. Driemeier).

https://doi.org/10.1016/j.neunet.2025.107763
Received 19 September 2024; Received in revised form 28 March 2025; Accepted 11 June 2025
Available online 26 June 2025
0893-6080/© 2025 Elsevier Ltd. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

explored the accuracy and suitability for real-time applications of a fully quantized ultra-lightweight object detection network.

Although quantization can significantly decrease computational cost and memory usage, the information loss can potentially lower overall model performance, particularly in terms of accuracy. The trade-off between quantization and model performance is a balance between achieving greater efficiency and maintaining acceptable levels of accuracy. Most algorithms used for quantizing parameters of AI models require a pre-trained model with at least 32-bit precision. Once a pre-trained model is obtained, techniques such as *Post-Training Quantization* (Gong et al., 2024; Liu et al., 2021) and *Quantization Aware Training* (de Lima & Carro, 2022; Kirtas et al., 2022; Siddegowda et al., 2022) are applied to ensure effective quantization. However, the lower the quantization, meaning the fewer bits used, the more degraded the model's performance becomes. Consequently, these methods typically do not use fewer than 8 bits to represent parameters after quantization. This limitation results in minimal reduction of the memory required to store the model and provides little significant improvement in computational efficiency. In Yang et al. (2020), a methodology to train ResNet models with full 8-bit integers is presented.

In Chu et al. (2021) the authors address the trade-off between model performance and computational efficiency by adopting a mixed-precision approach. Their proposal is based on the fact that different layers within neural networks contribute differently to overall performance and vary in their sensitivity to quantization.

Binarization is a 1-bit quantization where data can only have -1 or $+1$ values, and the ideia of *Binarized Neural Networks* open the possibility of a new era of more efficient neural network models that require less memory and can be applied to various types of problem (Qin et al., 2020). The works by Courbariaux et al. (2016) and Hubara et al. (2022) introduced an efficient method for training BNNs involving binary weights and activations. During the forward pass, weights and activations are binarized to either -1 or $+1$, reducing memory usage and enabling faster bitwise operations to replace most arithmetic computations. However, binary functions lack gradients necessary for backpropagation during training. To address this, real-valued weights are utilized to compute gradients and update parameters, facilitated by a hard hyperbolic tangent function that provides continuous and differentiable gradients. This approach ensures effective propagation of gradients through the network during backpropagation, enhancing training efficiency. Real-valued weights are subsequently binarized for use in the forward pass, maintaining computational efficiency during forward and inference while providing accurate gradient computation and parameter updates in the training phase. Furthermore, the authors devised a binary matrix multiplication GPU kernel to accelerate execution of the binary network compared to using an unoptimized GPU kernel. However, the reported findings are confined to a single model with convolutional layers and do not address scalability or extension of their proposals.

At the same time, Rastegari et al. (2016) studied two different approaches on large-scale datasets like ImageNet: the traditional BNNs with all weight values are approximated using binary values; and XNOR-Networks, that extends the first concept by also approximating inputs with binary values. The authors claimed that the last approach provided $\approx 58\times$ speed up and enabled the possibility of running the inference of state of the art deep neural network on CPU in real-time.

In 2018, Deng et al. (2018) introduced a novel approach for training DNNs. They addressed memory and computation bottlenecks by proposing a method to backpropagate through discrete activations and eliminate full-precision hidden weights during training. The authors constrained weights and activations in the ternary space $-1, 0, +1$ to form what they called gated XNOR networks.

In Wang et al. (2023), the authors introduced a novel approach for large language models, training the model from scratch with quantization, diverging from the common practice of applying quantization post-training. They developed BitNet, which is a 1-bit transformer architecture designed for large language models (llmS). BitNet employs low-precision binary weights and quantized activations, while maintaining high precision for the optimizer states and gradients during training. The results from (Wang et al., 2023) demonstrate that BitNet achieves competitive performance while substantially reducing memory usage compared to 8-bit quantization methods and 16-bit precision transformers.

In Ignatov and Ignatov (2020), Peng and Chen (2019) and Tang et al. (2020), different network binarization approaches were proposed to solve the lower prediction accuracy by using binary weights and fast bitwise operations. More recently, Ma et al. (2024) introduced a low-bit variant of llm, named BitNet 1.58, where each model parameter is ternary $-1, 0, +1$. This model achieves equivalent performance to models with the same number of parameters but trained with 16-bit precision, using the same number of tokens. Thus, BitNet 1.58 offers significant advantages including lower latency, reduced memory footprint, and decreased energy consumption. In Ma et al. (2024), the authors suggested that the 1.58-bit LLM model sets a new benchmark and paradigm for training next-generation high-performance and cost-effective LLMs.

The previous contributions create opportunities for more efficient neural network models that require less memory and can be applied to various types of problems. Additionally, these advancements support the development of specialized hardware optimized for low-bit neural networks.

The present study investigates the bit precision necessary for model parameters to achieve performance comparable to models using 32-bit resolution parameters, focusing on multiclass object classification in images. By exploring various quantization levels, from 1 bit to 4.08 bits, the study assesses how different degrees of numerical approximation impact neural network performance. Lower resolutions, such as 1-bit and 1.5-bit quantization, significantly reduce memory usage and computation but may compromise accuracy. As precision increases, a balance emerges between efficiency and performance, with finer quantization levels retaining more information while still optimizing computational resources. The models considered include those with fully connected layers (FCNN), convolutional layers (CVNN), and transformer blocks (Visual Transformer - VIT). Through comprehensive analyses, the study identified advanced quantization strategies that effectively maintain high classification accuracy while significantly enhancing computational efficiency. These strategies are particularly well-suited for deployment in resource-constrained environments, such as edge computing and embedded systems, where balancing performance and efficiency is critical. Section 2 outlines the parameter quantization method employed, Section 3 presents the data and training parameters, and Sections 4–6 respectively detail the FCNN, CVNN, and VIT models, along with comparative results across different parameter resolutions. Section 7 presents a summary of the results and a brief discussion of the performance of the models. Section 8 introduces a method for storing weights using fewer bits of resolution. Finally, Section 9 summarizes the conclusions drawn from this study and suggests some future steps to improve the results and conclusions of this work.

2. Quantization method

Since the model weights are the parameters that demand the most memory and computational resources in neural networks — both in convolutional layers and fully connected and attention layers — only the weights of the connections are constrained to use low-resolution parameters in the models. The biases of all layers retain a 32-bit resolution.

During training, the connection weights are stored with 32-bit resolution, but the layer activations are calculated using weights at the specified bit resolution. Therefore, a quantization method is required for the weights during the training process. The desired number of discrete values is first selected, for example:

1-bit resolution the model weights are constrained to 2 values:

$-1; +1$

1.5-bit resolution the model weights are constrained to 3 values:

$-1; 0; +1;$

2-bit resolution the model weights are constrained to 4 values:

$-1; -0,3333; +0.3333; +1;$

2.32-bit resolution the model weights are constrained to 5 values:

$-1; -0.5; 0; +0.5; +1;$

and so on.

It is important to note that, to reduce the memory required for storing model weights, they can be represented as positive integers, with their bit resolution determined by the defined number of possible discrete values. Weight quantization to the desired resolution is performed layer by layer in the model. Eqs. (1) to (4) implement the parameter quantization.

$$v_{max} = \frac{N_{values} - 1}{2} \tag{1}$$

$$\mathbf{W}_{norm} = \frac{\mathbf{W}}{\beta \bar{W}} \tag{2}$$

$$\mathbf{W}_q = \frac{\mathtt{round}\left(\mathbf{W}_{norm} v_{max} + v_{max}\right) - v_{max}}{v_{max}} \tag{3}$$

$$\mathbf{W}_q = \begin{cases} +1, & \text{if } \mathbf{W}_q > +1 - \frac{1}{N_{values}} \\ -1, & \text{if } \mathbf{W}_q < -1 + \frac{1}{N_{values}} \end{cases} \tag{4}$$

where N_{values} is the number of desired values for the layer weights, $\mathbf{W}$ is the tensor of weights of the layer, $\bar{W}$ is the mean of the layer weights, $\mathbf{W}_{norm}$ is the tensor of weights normalized by the mean, $1 \leq \beta \leq 2$ is a parameter for regulating the distribution of quantized weight values, $\mathbf{W}_q$ is the tensor of quantized weights, and `round` is a function performing rounding operation. The value of β in this work is set to 1.4 because this value ensures a uniform distribution among the three weight values when N_{values} equals 3.

As previously mentioned, during model training, weights are maintained as 32-bit real values. However, in the forward propagation calculations, quantized weights are employed, computed according to Eqs. (1) to (4). This approach is essential because if weights were stored in their quantized form, during training the parameter updates would be eliminated by the quantization function, hindering learning. Note that, most updates during training are typically on the order of 10^{-4} to 10^{-2}, which would round to zero in the quantization process, preventing the original parameters from being updated and thus hindering the model's learning. To address this issue, we employ a technique inspired by the implementation of Vector Quantized-Variational AutoEncoder (VQ-VAE) proposed by van den Oord et al. (2017). This method, which ensures effective parameter updates despite quantization, is detailed in Section 4.

3. Data and training parameters

Analyzing the number of bits required for a model to achieve the performance of models with 32-bit resolution weights constitutes a comparative study where the dataset used does not significantly influence the conclusions. Therefore, the CIFAR-10 dataset (Krizhevsky, 2009) is employed. This dataset features a straightforward multiclass classification task with low-resolution color images ($32 \times 32 \times 3$ pixels) across 10 object classes. It is split into a training set with 50,000 images and a test set with 10,000 images.

A critical aspect of AI models is their generalization capability. Training with original data often leads to overfitting. To evaluate the generalization capacity of models with low-resolution weights, data augmentation is also used during training. Minor transformations are applied to the images to avoid significant distortion, including the following:

- horizontal shift: $\pm 10\%$;
- vertical shift: $\pm 10\%$;
- zoom in/out: $\pm 20\%$;
- horizontal flip (left/right);
- rotation: $\pm 5°$.

In addition to using data augmentation to mitigate overfitting, models were trained with dropout layers. However, the inclusion of dropout was found to be ineffective in reducing overfitting, so results from models with dropout are not presented. The hyperparameters used in training the models are listed in Table 1.

It is observed that a considerable number of training epochs are needed for the cost function to completely converge during training. Additionally, models with transformer blocks require a greater number of epochs to achieve convergence.

To verify whether the number of parameters of the models influences training stability and performance, two models of different complexity and total number of parameters are configured and trained for each model type (models with only fully connected layers, models with convolutional layers, and models with transformer blocks).

4. Models with only fully connected layers

Algorithm 1 illustrates the forward propagation process in a fully connected layer with low-resolution connection weights. In Appendix A, the DenseBit class is presented, implementing a custom fully connected layer in TensorFlow/Keras that applies weight quantization to constrain parameter precision. The function `quant` performs the weight quantization defined by Eqs. (1) to (4), `no_gradient` is a placeholder function that prevents gradient calculation for its argument during model training, `activation` represents the chosen activation function for the layer, and b is the bias vector of the layer.

The quantized weights used to compute layer activations remain consistent during both training and inference. However, during training, when *trainable* variable is set to True, the calculated gradients and corresponding weight adjustments are stored in the normalized weight matrix (W_{norm}), ensuring no information is lost. This method allows for weight updates during training without sacrificing precision. The technique of adjusting parameters without information loss while using quantized weights in forward propagation calculations was adapted from Alcorn (2023).

4.1. Configuration of the models with fully connected layers

Two simple models are configured with differences in the number of layers and units per layer. Algorithm 2 outlines the forward propagation process for the simpler fully connected layers model (FCNN1), while Algorithm 3 presents the more complex model with a larger number of parameters (FCNN2).

In Algorithms 2 and 3, FCL represents the fully connected layer defined in Algorithm 1, and `Flatten` is a function that converts an image into a vector. The number of units in the output layers is set to 10 due to the presence of 10 object classes. Given the multiclass classification problem, the output layer employs a `softmax` activation function, while `relu` is used in other layers. The number of values used for each weight, which is proportional to the number of bits, is determined by the parameter N_{values}. In both models, weight connections and layer biases are initialized using standard methods: *Glorot Uniform* for weights and zeros for biases. No regularization techniques or parameter constraints are applied.

The number of units used in the layers of the simpler model (FCNN1) are respectively 512, 256, 128, and 10, totaling 1,738,890 weights and biases. In the more complex model (FCNN2), the numbers of units in the layers are respectively 512, 256, 128, and 10, totaling 3,837,066 weights and biases.

Table 1
Hyperparameters used in model training.

Hyperparameter	Value
Cost function	Categorical Cross Entropy
Optimization method	Stochastic Gradient Descendent with Momentum
Momentum rate	0.92
Metrics	Accuracy
Learning rate for fully connected and convolutional models	0.001 (fixed)
Learning rate for models with transformer blocks	0.01 (fixed)
Batch size	256
Number of epochs with original data for fully connected and convolutional models	200
Number of epochs with data augmentation for fully connected and convolutional models	1000
Number of epochs with original data for models with transformer blocks	300
Number of epochs with data augmentation for models with transformer blocks	2000

Algorithm 1 Forward propagation calculation process in a fully connected layer with low-resolution weights.

Require: Input $\mathbf{x}$, weights $\mathbf{W}$, bias $\mathbf{b}$, scaling factor β, weight mean $\bar{W}$, trainable flag *trainable*
Ensure: Activations $\mathbf{a}$
1: Calculate weights adjust factor: $\gamma = \beta\bar{W}$
2: Calculate normalized weights: $\mathbf{W}_{norm} = \frac{\mathbf{W}}{\gamma}$
3: **if** *trainable* **then**
4: Quantize weights for training: $\mathbf{W}_q = \mathbf{W}_{norm} + \texttt{no_gradient}(\texttt{quant}(\mathbf{W}_{norm}) - \mathbf{W}_{norm})$
5: **else**
6: Quantize weights for inference: $\mathbf{W}_q = \texttt{quant}(\mathbf{W}_{norm})$
7: **end if**
8: Calculate activations: $\mathbf{a} = \texttt{activation}(\gamma\mathbf{W}_q \cdot \mathbf{x} + \mathbf{b})$
9: **return** $\mathbf{a}$

Algorithm 2 Forward propagation process in the simpler fully connected layers model (FCNN1).

Require: Input image $\mathbf{x}$, number of weight values N_{values}
Ensure: Predicted output **ypred**
1: $\mathbf{xf} = \texttt{Flatten}(\mathbf{x})$
2: $\mathbf{a1} = \text{FCL}(\text{units} = 512, N_{values}, \text{activation} = \texttt{relu})(\mathbf{xf})$
3: $\mathbf{a2} = \text{FCL}(\text{units} = 256, N_{values}, \text{activation} = \texttt{relu})(\mathbf{a1})$
4: $\mathbf{a3} = \text{FCL}(\text{units} = 128, N_{values}, \text{activation} = \texttt{relu})(\mathbf{a2})$
5: $\mathbf{ypred} = \text{FCL}(\text{units} = 10, N_{values}, \text{activation} = \texttt{softmax})(\mathbf{a3})$

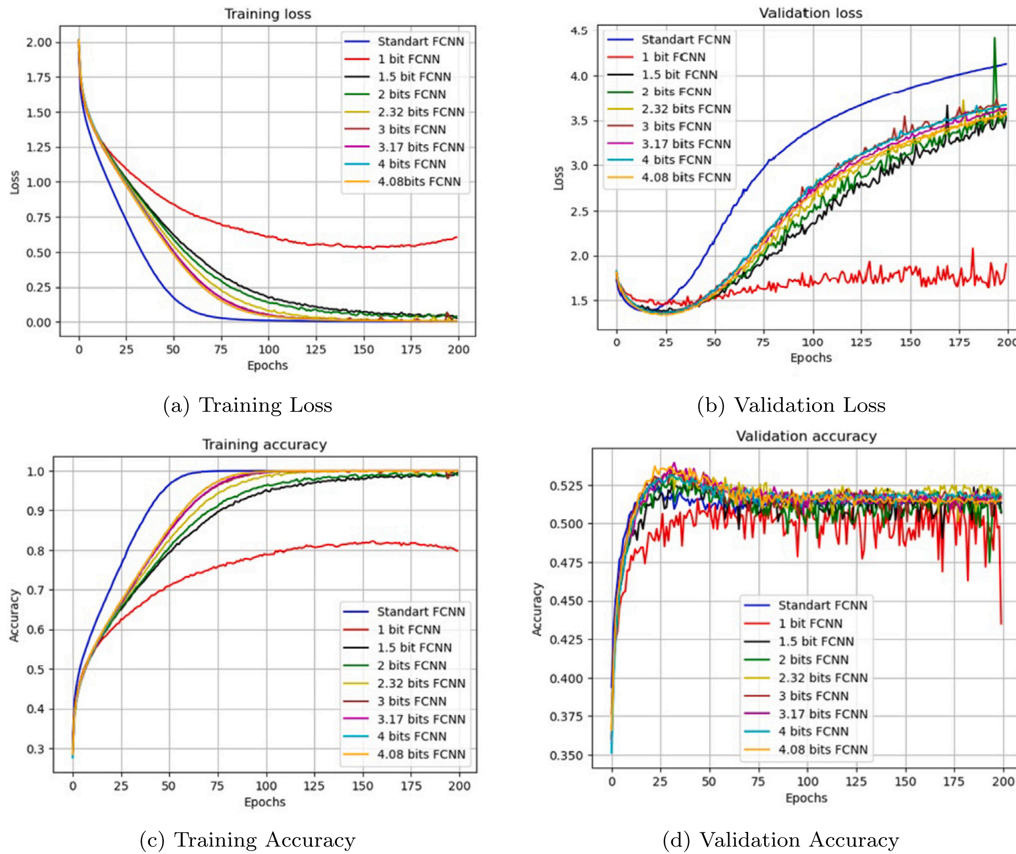
Algorithm 3 Forward propagation process in the more complex fully connected layers model (FCNN2).

Require: Input image $\mathbf{x}$, number of weight values N_{values}
Ensure: Predicted output **ypred**
1: $\mathbf{xf} = \texttt{Flatten}(\mathbf{x})$
2: $\mathbf{a1} = \text{FCL}(\text{units} = 1024, N_{values}, \text{activation} = \texttt{relu})(\mathbf{xf})$
3: $\mathbf{a2} = \text{FCL}(\text{units} = 512, N_{values}, \text{activation} = \texttt{relu})(\mathbf{a1})$
4: $\mathbf{a3} = \text{FCL}(\text{units} = 256, N_{values}, \text{activation} = \texttt{relu})(\mathbf{a2})$
5: $\mathbf{a4} = \text{FCL}(\text{units} = 128, N_{values}, \text{activation} = \texttt{relu})(\mathbf{a3})$
6: $\mathbf{ypred} = \text{FCL}(\text{units} = 10, N_{values}, \text{activation} = \texttt{softmax})(\mathbf{a4})$

4.2. The results of the models with fully connected layers

We utilize models with low-resolution weights set to $2, 3, 4, 5, 8, 9, 16$, and 17 different values, corresponding to resolutions of $1, 1.5, 2, 2.32, 3, 3.17, 4$, and 4.08 bits per weight, respectively. It is important to clarify that the reported results are examples from individual training runs. However, extensive repetitions were conducted for each model under identical conditions, consistently producing similar outcomes. Therefore, presenting a single example result for each model is highly indicative.

Fig. 1 shows the training results for the models featuring only fully connected layers with the simpler configuration (FCNN1), while Fig. 2



(a) Training Loss

(b) Validation Loss

(c) Training Accuracy

(d) Validation Accuracy

Fig. 1. Training and validation results for the simpler models with only fully connected layers (FCNN1) for various resolutions used in the weights.

presents the results for the more complex models with various resolutions used for the weights. The results of a standard 32-bit parameter network is included as a benchmark reference.

The results depicted in Figs. 1 and 2 reveal several crucial findings:

1. Training the simpler model (FCNN1) with 1-bit weights (N_{values} = 2) proves ineffective, resulting in unsatisfactory performance.
2. Models with lower-resolution parameters require more epochs to achieve adequate training.
3. Models with a higher number of parameters in the low-resolution configuration progressively approach the performance level of the standard model.
4. All models, including the standard model, demonstrate overfitting issues, leading to inferior performance on validation data compared to training data.

Figs. 3 and 4 illustrate the training outcomes for both simpler (FCNN1) and more complex (FCNN2) models with data augmentation over 1000 training epochs. The specific image transformations used during training are detailed in Section 3.

It is important to note that multiple training runs were conducted for all models, consistently yielding similar results.

Upon analyzing the training results of the fully connected neural network models with data augmentation, as depicted in Figs. 3 and 4, several insights emerge:

1. Both models utilizing 1-bit weights exhibit training instability and produce unsatisfactory results.
2. Interestingly, the 1.5-bit model performs better than the 2-bit model despite the latter having higher resolution, suggesting that the inclusion of zero among possible weight values plays a crucial role.
3. The more complex model with 2-bit weights achieves satisfactory results and performs comparably to models with higher-resolution weights, indicating that a large number of parameters allows effective learning even with lower-resolution weights.
4. Similar to training without data augmentation, models with lower-resolution weights require a greater number of epochs to achieve results comparable to those of standard 32-bit models.

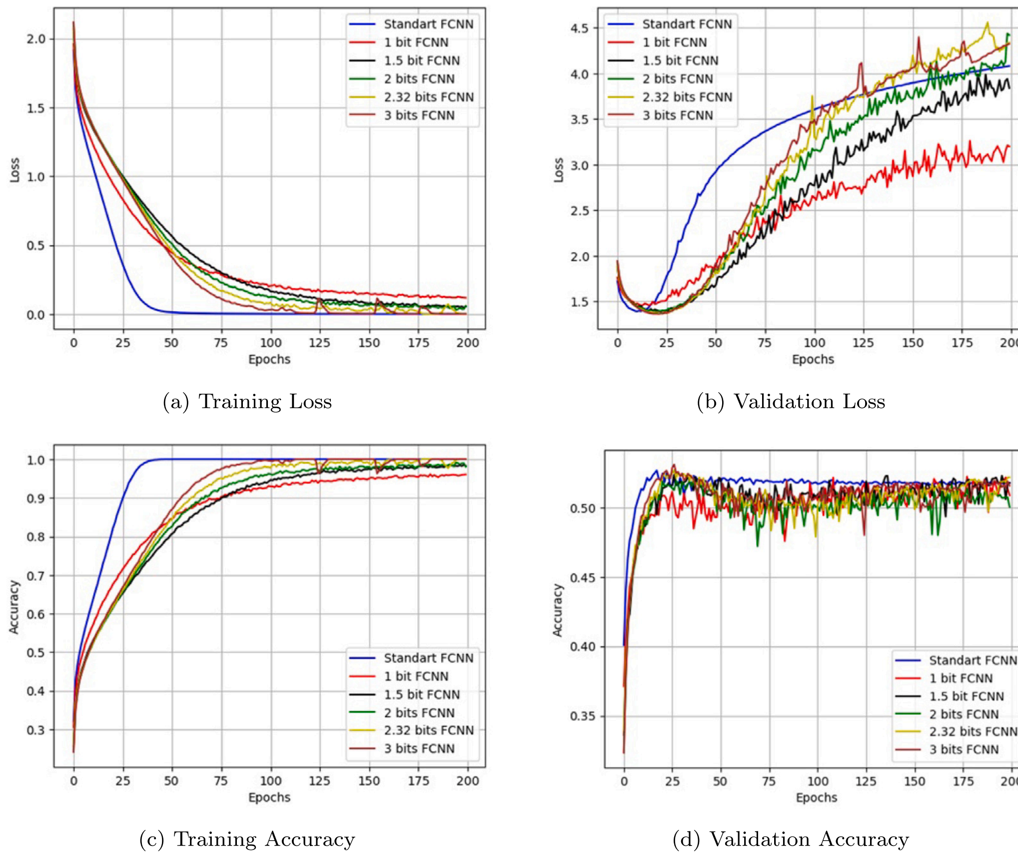
Note that all models that do not exhibit training instability show overfitting, albeit to a lesser degree than observed in training without data augmentation. Therefore, this shows that models with low resolution weights are capable of generalization in the same way as 32-bit models.

5. Models with convolutional layers

The only difference between the convolutional layer and the fully connected layer with low-resolution weights is that in computing the activations, convolution operation is used between the filters (connection weights) and the input tensor of the layer, rather than matrix multiplication. Eq. (5) performs convolution operation in calculating activations for convolutional layers with low resolution weights.

$$\mathbf{a} = \mathtt{activation}\left(\mathtt{conv}\left(\gamma \mathbf{W}_q, \mathbf{x}\right) + \mathbf{b}\right) \tag{5}$$

where $\mathtt{conv}\left(\gamma \mathbf{W}_q, \mathbf{x}\right)$ performs two dimensional convolution of $\mathbf{x}$ by $\gamma \mathbf{W}_q$. It should be noted that Eq. (5) replaces the activation calculation in Algorithm 1, which implements the forward propagation process in fully connected layers. In Appendix B, the Conv2DBit class is presented, implementing a custom 2D convolutional layer in TensorFlow/Keras that applies weight quantization to constrain parameter precision.



(a) Training Loss (b) Validation Loss

(c) Training Accuracy (d) Validation Accuracy

Fig. 2. Training and validation results for the more complex models with only fully connected layers (FCNN2) for various resolutions used in the weights.

Algorithm 4 Forward propagation process for the simpler convolutional layer model (CVNN1).

Require: Input image $\mathbf{x}$, number of weight Values N_{values}
Ensure: Predicted output $\mathbf{ypred}$

1: $\mathbf{a1}$ = Conv2D(units = 64, N_{values}, activation = `relu`, padding = `same`)($\mathbf{x}$)
2: $\mathbf{a1}$ = MaxPool2D((2, 2), strides = (2, 2))($\mathbf{a1}$)
3: $\mathbf{a2}$ = Conv2D(units = 128, N_{values}, activation = `relu`, padding = `same`)($\mathbf{a1}$)
4: $\mathbf{a2}$ = MaxPool2D((2, 2), strides = (2, 2))($\mathbf{a2}$)
5: $\mathbf{a3}$ = Conv2D(units = 256, N_{values}, activation = `relu`, padding = `same`)($\mathbf{a2}$)
6: $\mathbf{a3}$ = MaxPool2D((2, 2), strides = (2, 2))($\mathbf{a3}$)
7: $\mathbf{a4}$ = Flatten($\mathbf{a3}$)
8: $\mathbf{a5}$ = FCL(units = 128, N_{values}, activation = `relu`)($\mathbf{a4}$)
9: $\mathbf{ypred}$ = FCL(units = 10, N_{values}, activation = `softmax`)($\mathbf{a5}$)

5.1. Configuration of the models with convolutional layers

Two models are configured with differences in the number of layers and units per layer. Algorithm 4 outlines the forward propagation calculation for the simpler convolutional layer model (CVNN1), while Algorithm 5 details the calculation process for the more complex model (CVNN2).

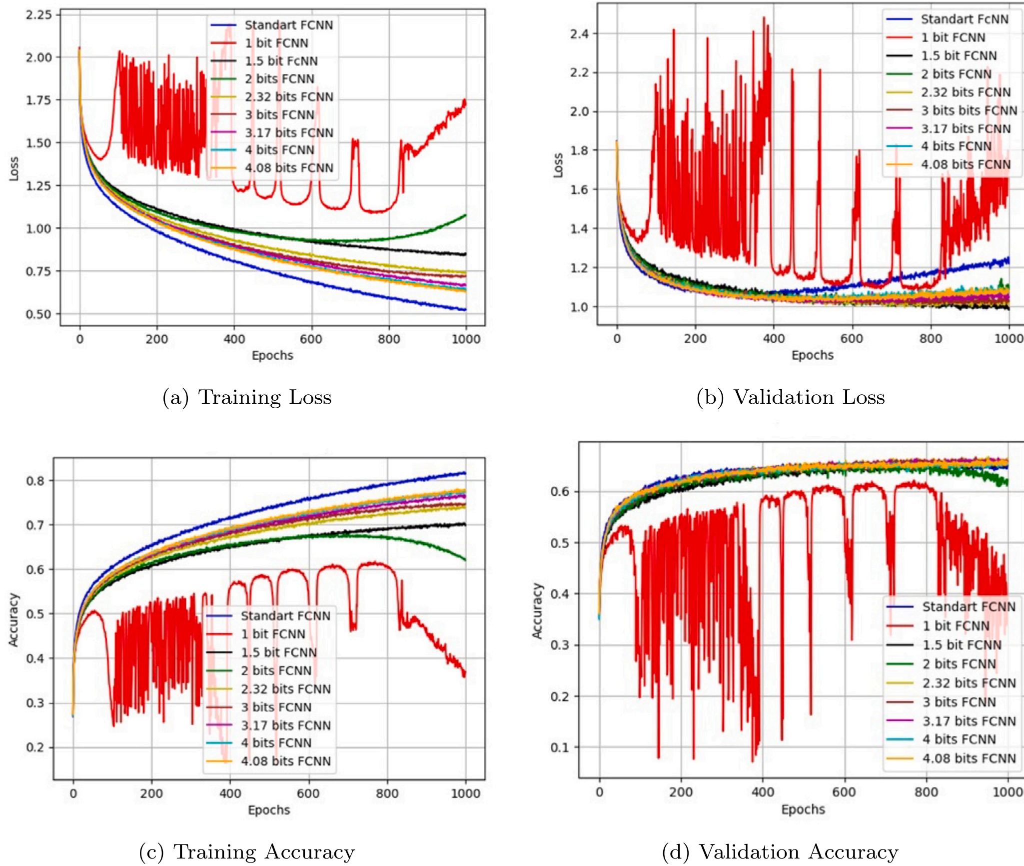
In Algorithms 4 and 5, Conv2D denotes a two-dimensional convolutional layer, while MaxPool2D signifies a two-dimensional max-pooling layer. Each convolutional layer employs 3 × 3 filters with a `relu` activation function, a stride of 1, and padding to preserve the input tensor dimensions in the output tensors. The first fully connected layer also uses `relu` activation, whereas the output layer adopts `softmax` activation. The parameter N_{values} defines the number of possible values for the connection weights.

In both models, connection weights and biases are initialized using standard methods: *Glorot Uniform* for weights and zeros for biases. No regularization techniques or parameter constraints are applied. The simpler model (CVNN1) comprises 896,522 parameters, while the more complex model (CVNN2) includes 8,776,330 parameters, accounting for both weights and biases.

5.2. Results obtained with the models with convolutional layers

In Fig. 5, the training results are displayed for the simpler convolutional layer models (CVNN1), while Fig. 6 presents the results for the more complex models (CVNN2), both for various resolutions used in the layer weights. Once again, results from standard networks with 32-bit parameters are included as a benchmark for the desired performance. It is important to note that multiple training tests were conducted for all models, and all results are very similar.

Analyzing the training results of the convolutional models shown in Figs. 5 and 6, one can observe that the results of the low-resolution models at the end of training, except for the simpler 1-bit model (CVNN1), are nearly identical to those of the standard models. Other observations can be done:



(a) Training Loss (b) Validation Loss

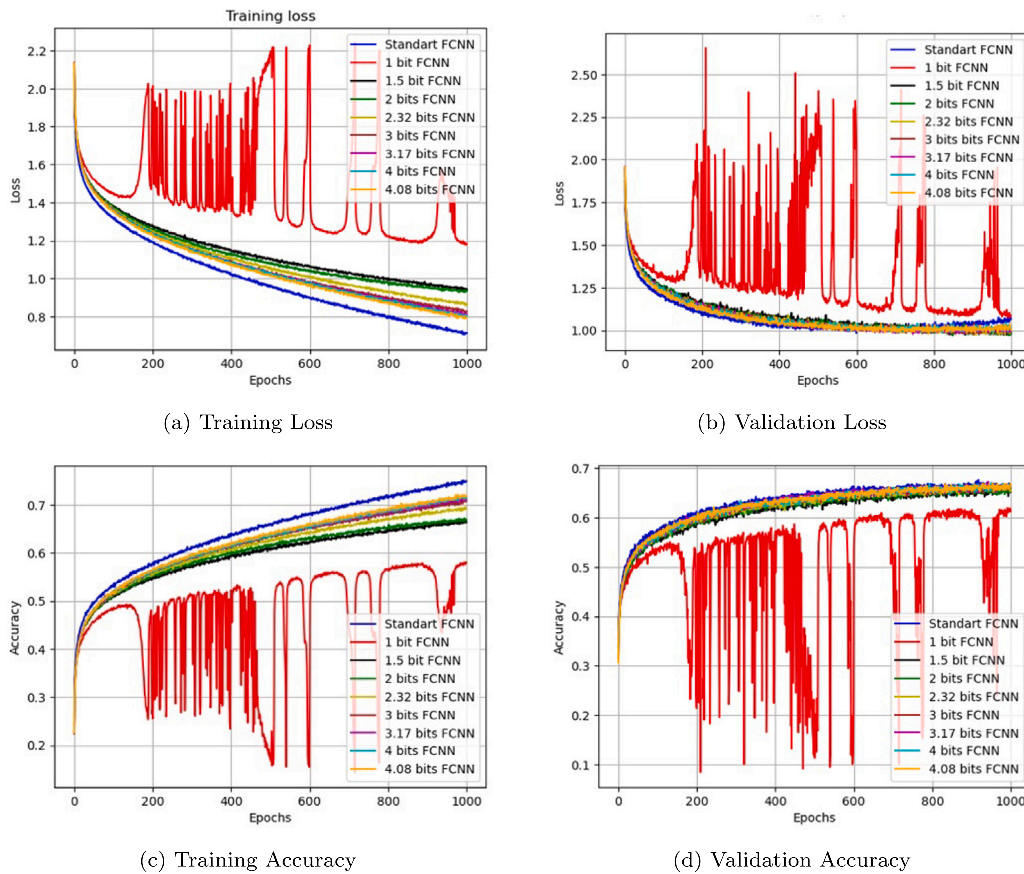
(c) Training Accuracy (d) Validation Accuracy

Fig. 3. Training and validation results for the simpler models with only fully connected layers (FCNN1) using data augmentation for various resolutions used in the weights.

Algorithm 5 Forward propagation process for the more complex convolutional layer model (CVNN2).

Require: Input image $\mathbf{x}$, number of weight values $\mathbf{N_{values}}$
Ensure: Predicted output **ypred**
1: $\mathbf{a1} = \text{Conv2D}(\text{units} = 128, N_{values}, \text{activation} = \texttt{relu}, \text{padding} = \texttt{same})(\mathbf{x})$
2: $\mathbf{a1} = \text{Conv2D}(\text{units} = 128, N_{values}, \text{activation} = \texttt{relu}, \text{padding} = \texttt{same})(\mathbf{a1})$
3: $\mathbf{a1} = \text{MaxPool2D}((2,2), \text{strides} = (2,2))(\mathbf{a1})$
4: $\mathbf{a2} = \text{Conv2D}(\text{units} = 256, N_{values}, \text{activation} = \texttt{relu}, \text{padding} = \texttt{same})(\mathbf{a1})$
5: $\mathbf{a2} = \text{Conv2D}(\text{units} = 256, N_{values}, \text{activation} = \texttt{relu}, \text{padding} = \texttt{same})(\mathbf{a2})$
6: $\mathbf{a2} = \text{MaxPool2D}((2,2), \text{strides} = (2,2))(\mathbf{a2})$
7: $\mathbf{a3} = \text{Conv2D}(\text{units} = 512, N_{values}, \text{activation} = \texttt{relu}, \text{padding} = \texttt{same})(\mathbf{a2})$
8: $\mathbf{a3} = \text{Conv2D}(\text{units} = 512, N_{values}, \text{activation} = \texttt{relu}, \text{padding} = \texttt{same})(\mathbf{a3})$
9: $\mathbf{a3} = \text{MaxPool2D}((2,2), \text{strides} = (2,2))(\mathbf{a3})$
10: $\mathbf{a4} = \text{Flatten}(\mathbf{a3})$
11: $\mathbf{a5} = \text{FCL}(\text{units} = 512, N_{values}, \text{activation} = \texttt{relu})(\mathbf{a4})$
12: $\mathbf{ypred} = \text{FCL}(\text{units} = 10, N_{values}, \text{activation} = \texttt{softmax})(\mathbf{a5})$

- The simpler model (CVNN1) with 1-bit weights exhibits instability throughout all training processes and fails to learn the data, unlike the more complex 1-bit model, which despite some instability during training, is able to learn effectively;
- The simpler models (CVNN1) with 1.5 and 2-bit resolutions require more epochs to achieve comparable results compared to the standard 32-bit model. Conversely, models with resolutions of 2.32, 3, 3.17, 4, and 4.08 bits among the simpler CVNN1 models require fewer training epochs than the standard model. Also, the low-resolution more complex models (CVNN2) with more than 1-bit require the same number of epochs to train as the standard model;
- The simpler low-resolution models (CVNN1) exhibit occasional sharp fluctuations in the cost function and accuracy, which quickly stabilize. These fluctuations are likely associated with simultaneous changes in a large number of weights. Notably, these oscillations do not occur during training of the more complex models (CVNN2).



(a) Training Loss (b) Validation Loss

(c) Training Accuracy (d) Validation Accuracy

Fig. 4. Training and validation results for the more complex models with only fully connected layers (FCNN2) using data augmentation for various resolutions used in the weights.

Excluding the simpler 1-bit model (CVNN1), all other models demonstrate overfitting problems. To verify the generalization ability of low-resolution convolutional models, training is repeated with data augmentation. The image transformations applied are identical to those used in the training of the models with only fully connected layers. Fig. 7 presents the training results for the simpler models (CVNN1), while Fig. 8 displays the results for the more complex models (CVNN2), using 1000 training epochs. Multiple training runs were conducted for all models, and consistent results were observed across all experiments.

Analyzing the results presented in Figs. 7 and 8 reveals that data augmentation tends to induce instability in the training of low-resolution weight models. All the low resolutions models with 1, 2 and 3 bits exhibit training instability and unsatisfactory results, thus none of these models trained with data augmentation yield satisfactory outcomes. Furthermore, except for the models experiencing instability, the simpler low-resolution models (CVNN1) generally show slightly worse results compared to the standard model, whereas the low-resolution more complex models (CVNN2) exhibit results similar to the standard model. An important observation is that models incorporating zero as a possible weight value, specifically the 1.5-bit, 2.32-bit, 3.17-bit, and 4.08-bit models, outperform models where zero is not included among the possible weight values.

To allow a better analysis of low-resolution convolutional layers, models with 5×5 filters are trained to investigate whether using larger filters impacts the performance of the model. It is important to note that these models are identical to the previously analyzed convolutional models, with the only modification being the filter size. Fig. 9 shows the training results with data augmentation for the more complex models using 5×5 filters, employing 1000 training epochs. Multiple training sessions were conducted for all models, yielding consistent results across all the experiments.

The results presented in Fig. 9 shows that using 5×5 filters increases the issue of instability. In this case, aside from the more complex models (CVNN2) with 1-bit and 2-bit resolutions, the 3-bit model also exhibits instability. This result reinforces the observation that models with an even number of possible weight values tend to encounter more training difficulties compared to models with an odd number of possible weight values. Note that models with an odd number of possible values include weights that can be zero, whereas models with even numbers of possible values do not. This finding underscores the importance of including weights with values of zero, particularly for low-resolution weights and models with a few number of parameters.

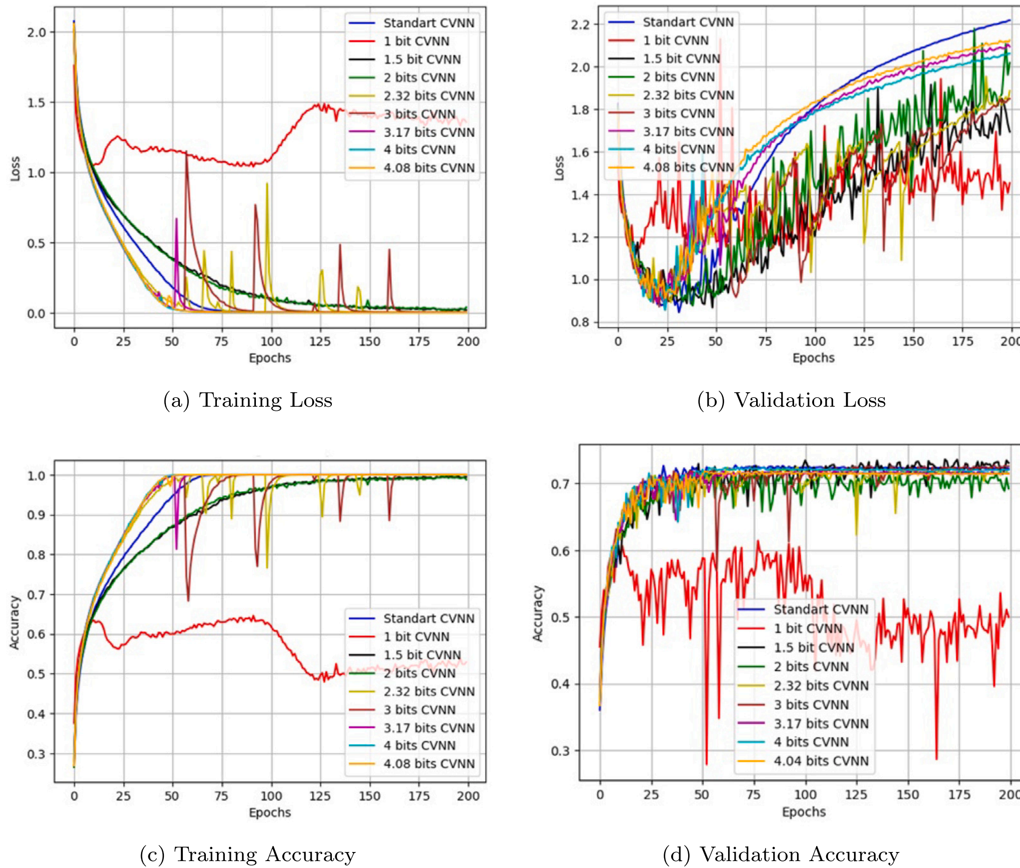
6. Models with transformer blocks (Visual Transformer - VIT)

A transformer block, including its attention mechanism, primarily consists of embedding layers, fully connected layers, and normalization layers. The fully connected layers of a transformer block with low-resolution weights are the same as those presented in Algorithm 1. Since the embedding and normalization layers have a small number of parameters compared to the fully connected layers, 32-bits parameters are used in these layers.

6.1. *Configuration of the visual transformer models*

Two models that differ only in the number of transformer blocks and the units in each fully connected layer are used in this comparative study. Algorithm 6 presents the forward propagation process in the VIT models.

The components of Algorithm 6 are detailed as follows:



(a) Training Loss (b) Validation Loss

(c) Training Accuracy (d) Validation Accuracy

Fig. 5. Training and validation results for the simpler models with Convolutional layer models (CVNN1) for various resolutions used in the weights.

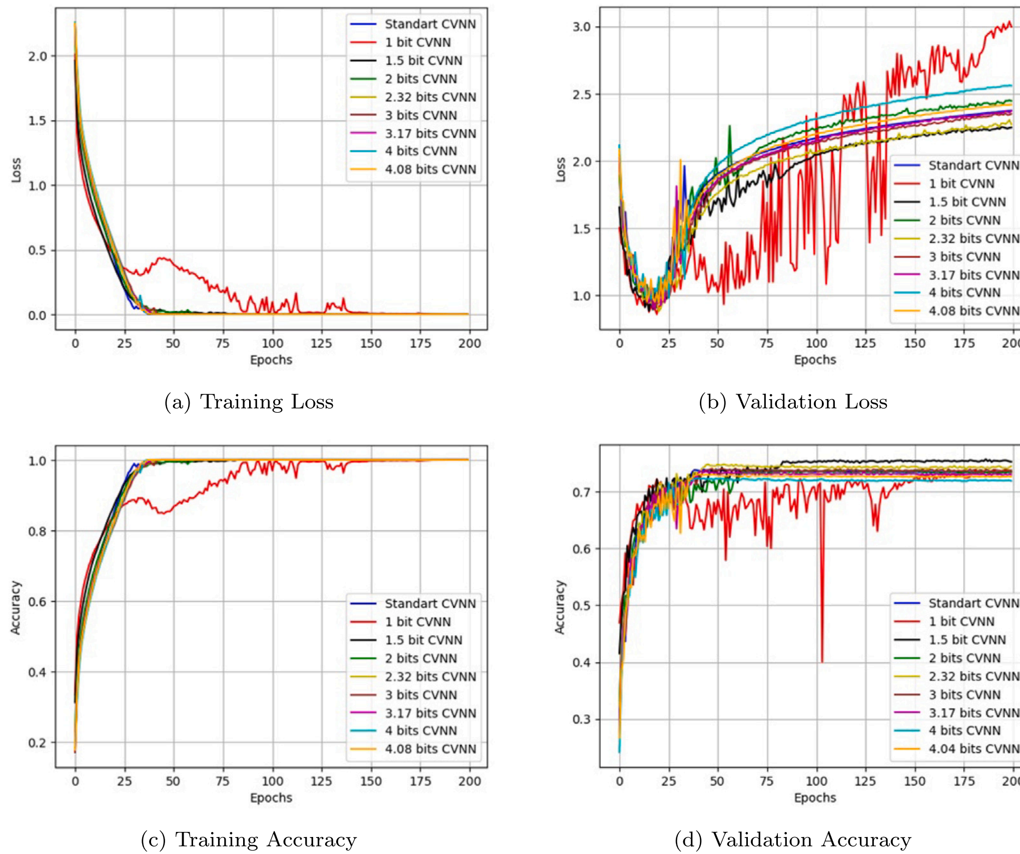
- **FLC** (Fully Connected Layer): This layer uses low-resolution weights, and its calculation process is described in Algorithm 1. As previously discussed, the number of possible values for the connection weights is determined by the parameter N_{values}. Additionally, the Gaussian Error Linear Unit (GELU) activation function is employed in certain layers to enhance non-linearity.
- `Patches`($patch_size$): This function generates image patches with dimensions ($batch_size, num_patches_h \times num_patches_w, patch_size \times patch_size \times channels$). Here, $batch_size$ represents the number of examples in a batch, $num_patches_h$ is the number of patches along the image height, $num_patches_w$ is the number of patches along the image width, $patch_size$ is the size of each patch in pixels, and $channels$ indicates the number of image channels.
- `PatchEncoder`($num_patches$, emb_dim): This is a standard function used in visual transformers that performs the embedding encoding of image patches along with their respective positional encodings. The $num_patches$ parameter denotes the total number of patches per image, while emb_dim represents the dimensionality of the embedding encoding for the patches. The encoding of image patches is carried out using a fully connected layer with a linear activation function and 32-bit weight resolution. Positional encoding is performed using a standard embedding layer.
- `LayerNormalization`: This is a standard layer that normalizes the features of each example, ensuring that the output remains stable and centered.
- `Dropout(dropout_rate)`: This is a standard dropout layer where `dropout_rate` specifies the fraction of the input units to drop during training to prevent overfitting.
- `Attention(emb_dim, num_heads, dropout_rate)`: This component represents the standard attention mechanism utilized in visual transformer models. The arguments include `emb_dim`, `dropout_rate`, and `num_heads`. The last argument indicates the number of attention heads. The calculation process for this attention mechanism is detailed in Algorithm 7. Within this algorithm, the `reshape` function resizes a tensor according to the specified dimensions, while the `permute` function rearranges the axes of a tensor based on the provided order. All other terms have been defined in earlier sections.

In Table 2, the hyperparameters used in the two VIT models implemented in this study are presented. For both models, the weights of the connections and biases of the fully connected layers are initialized using the standard method, i.e., *Glorot Uniform* initialization for weights and zeros for the biases where applicable. No regularization techniques are used, and there are no parameter constraints. The total number of parameters for the simpler model (VIT1) is 4,766,282, while for the more complex model (VIT2) it is 10,573,770.

6.2. Results obtained with the VIT models

Fig. 10 shows the training results for the simpler VIT models (VIT1) while Fig. 11 presents the results for the more complex VIT models (VIT2) for several resolutions used in the weights. Once again, the results for standard models with 32-bit parameters are provided as a reference for the desired performance. Again, it is important to note that multiple training tests were conducted for all models, and all results are highly consistent.

The training results of the VIT models presented in Figs. 10 and 11, show that the models with 1-bit weights exhibit poorer results compared to other models, but they are capable of learning the data, however, all other models, both simpler (VIT1) and more complex (VIT2), show results similar to those of the standard 32-bit resolution



(a) Training Loss (b) Validation Loss

(c) Training Accuracy (d) Validation Accuracy

Fig. 6. Training and validation results for the more complex models with convolutional layer models (CVNN2) for various resolutions used in the weights.

Table 2
Hyperparameters used in model training.

Hyperparameter	Value
patch_size	4
num_patches (image_size/patch_size)^2	64
emb_dim	64
num_heads	4
transformer_units [$emb_dim \times 2, emb_dim$]	[128, 64]
transformer_layers	2 (simpler model) 4 (complex model)
mlp_head_units	[1024, 512] (simpler model) [2048, 1024] (complex model)
Loss function	Categorical Cross Entropy
Optimization method	Gradient Descent with Momentum

models. Also, for the VIT-type models, the parameter resolution appears to have less influence on the results quality compared to models with fully connected and convolutional layers.

All models exhibit overfitting problems regardless of the resolution of their weights. To verify the generalization capability of the low-resolution models, training is repeated with data augmentation. The image transformations applied are identical to those used in training models with fully connected and convolutional layers. Fig. 12 presents the training results for the simpler models (VIT1), while Fig. 13 presents the results for the more complex models (VIT2) with data augmentation across 2000 training epochs. It is noted that multiple trainings were conducted for all models and all results are consistent.

Results of Figs. 12 and 13 show that training with data augmentation introduces instability in the results of several low-resolution models. For the simpler models (VIT1), instability occurs in models with 1, 1.5, 2, 2.32, and 3-bit resolution weights. Among the more complex models, instability is observed in models with 1 and 2-bit resolution weights. For the low-resolution models that do not exhibit training instability, the results are very similar to those obtained with standard 32-bit models. Furthermore, as expected, training with data augmentation reduces the problem of overfitting but does not completely eliminate it, however, the overfitting behavior of the low resolution models is the same as the standard 32-bit models.

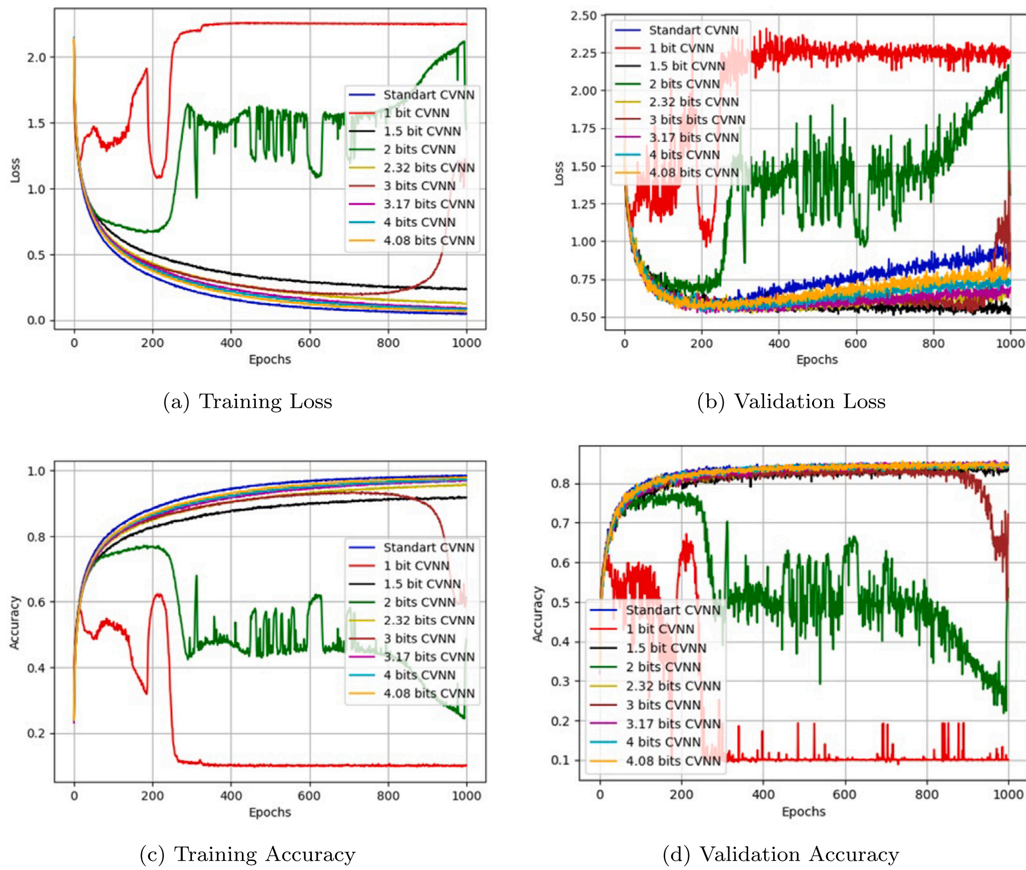
Comparing the results of the VIT models with models using convolutional layers, it is observed that both types of models yield similar results. However, the VIT models appear to have less training instability without data augmentation and more instability with data augmentation. Additionally, the VIT models require a larger number of epochs to achieve comparable results.

7. Summary of the results

The results for fully connected, convolutional, and transformer-based models are summarized, respectively, in Tables 3, 4, and 5, which presents median values calculated from several runs. The best and worst performance are highlighted in green and red, respectively.

FCNN models showed strong overfitting, reaching 100% training accuracy but only 50%–66% in validation without data augmentation. With augmentation, validation accuracy improved to 82%, and higher quantization levels (e.g., 4 bits) slightly enhanced generalization. However, FCNNs remained the least robust, struggling with generalization.

CVNN models exhibited more stable performance with reduced overfitting, achieving 72%–86% validation accuracy even without data augmentation. Lower quantization levels (1.5 and 2.32 bits) still performed well, reaching up to 87.5%. Data augmentation particularly improved CVNN2, reinforcing CNNs' resilience to quantization.



(a) Training Loss (b) Validation Loss

(c) Training Accuracy (d) Validation Accuracy

Fig. 7. Training and validation results for the simpler models with only fully connected layers (FCNN1) using data augmentation for various resolutions used in the weights.

Table 3
Summary of the results for the fully connected layer models (FCNN1, simpler, and FCNN2, more complex model).

Resolution	FCNN1 (No Aug)				FCNN1 (Aug)				FCNN2 (No Aug)				FCNN2 (Aug)			
	Train data		Val data		Train data		Val data		Train data		Val data		Train data		Val data	
	Loss	Acc	Loss	Acc	Loss	Acc	Loss	Acc	Loss	Acc	Loss	Acc	Loss	Acc	Loss	Acc
Standard	0.002	1.0	4.125	0.514	0.669	0.766	1.004	0.655	$< 10^{-3}$	1.0	4.080	0.518	0.406	0.861	1.068	0.662
1 bit	0.834	0.699	1.908	0.435	1.130	0.601	1.173	0.578	0.159	0.946	3.200	0.509	0.971	0.657	1.085	0.612
1.5 bits	0.035	0.991	3.523	0.507	0.910	0.683	1.032	0.634	0.048	0.985	3.840	0.518	0.694	0.757	0.980	0.658
2 bits	0.043	0.988	3.534	0.508	1.180	0.593	1.216	0.581	0.183	0.947	4.420	0.500	0.665	0.768	0.977	0.658
2.32 bits	0.007	1.0	3.59	0.518	0.820	0.710	1.013	0.642	0.001	1.0	4.335	0.522	0.6202	0.786	1.014	0.656
3 bits	0.005	1.0	3.627	0.518	0.805	0.716	1.010	0.646	0.001	1.0	4.324	0.516	0.551	0.811	1.0	0.662
3.17 bits	0.004	1.0	3.623	0.516	0.778	0.727	1.004	0.653	0.002	1.0	4.161	0.517	0.531	0.817	0.994	0.664
4 bits	0.004	1.0	3.668	0.519	0.758	0.734	0.999	0.650	0.002	1.0	4.341	0.510	0.521	0.824	1.010	0.666
4.08 bits	0.004	1.0	3.656	0.516	0.746	0.739	1.001	0.649	0.002	1.0	4.194	0.515	0.501	0.829	1.001	0.667

VIT models outperformed the others, with validation accuracy between 74% and 83%, generally surpassing FCNNs and CVNNs. They were also less sensitive to quantization, maintaining strong results even at 1.5 bits. Data augmentation further improved stability, with VIT2 at 2.32 bits achieving the best overall validation accuracy of 83%, highlighting the robustness and effectiveness of transformer-based models.

Note, however, that the comparison between the various types of models is not very appropriate because models with different numbers of parameters were used and this comparison is not objective of this work.

8. Memory reduction for models with low-resolution weights

The architecture of current computers does not facilitate efficient multiplications involving both integers and real numbers. Therefore, for low-resolution weight models to achieve greater computational efficiency, it is essential to develop optimized hardware capable of performing operations with low-bit integers. Nevertheless, even with existing hardware, low-resolution models offer a significant advantage by requiring substantially less memory.

For instance, consider a model with 1.5-bit weights, which can take on three possible values: -1, 0, and $+1$. In this case, there are 243 (or 3^5) possible combinations of five weights using these values. This implies that five weights of 1.5 bits can be stored within a single byte (8 bits). In comparison with storing weights in a 32-bit format (4 bytes), results in a memory reduction factor of 20.

Table 6 presents the memory reduction for each of the low-resolution models analyzed in this study compared to 32-bit weights, assuming a byte is the minimum memory unit. It is important to note that this memory reduction only considers the weights of the connections.

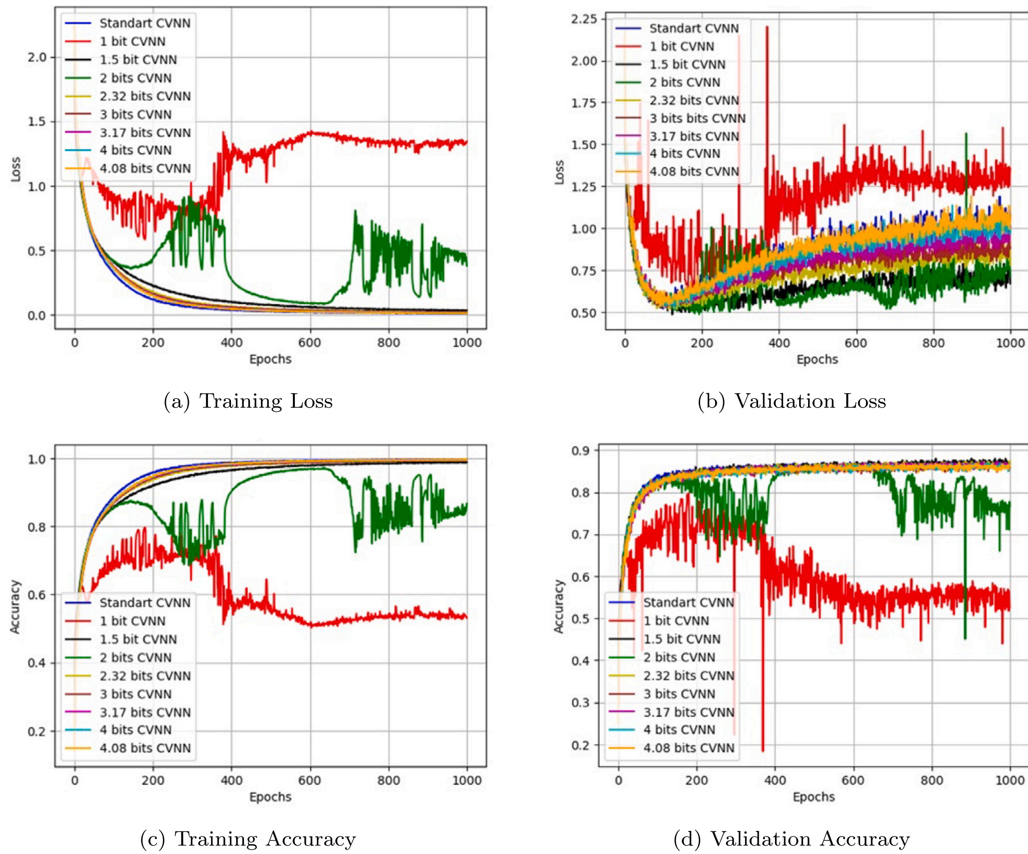


Fig. 8. Training and validation results for the MORE complex convolutional layer models (CVNN2) using data augmentation for various resolutions used in the weights.

Table 4
Summary of the results for the fully convolutional layer models (CVNN1, simpler, and CVNN2, more complex model).

Resolution	CVNN1 (No Aug)				CVNN1 (Aug)				CVNN2 (No Aug)				CVNN2 (Aug)			
	Train		Val		Train		Val		Train		Val		Train		Val	
	Loss	Acc	Loss	Acc	Loss	Acc	Loss	Acc	Loss	Acc	Loss	Acc	Loss	Acc	Loss	Acc
Standard	$< 10^{-3}$	1.0	2.217	0.722	0.029	0.990	0.200	0.844	$< 10^{-4}$	1.0	2.374	0.734	0.005	0.998	1.080	0.864
1 bit	1.380	0.531	1.452	0.500	2.258	0.100	2.259	0.100	$< 10^{-4}$	1.0	2.998	0.734	1.231	0.569	1.353	0.527
1.5 bits	0.015	0.997	1.693	0.731	0.123	0.959	0.542	0.837	$< 10^{-4}$	1.0	2.248	0.752	0.009	0.997	0.670	0.871
2 bits	0.060	0.980	2.017	0.693	1.303	0.536	1.308	0.534	$< 10^{-4}$	1.0	2.446	0.735	0.300	0.902	0.708	0.766
2.32 bits	0.003	1.0	1.885	0.715	0.065	0.978	0.662	0.842	$< 10^{-4}$	1.0	2.272	0.745	0.003	0.999	0.779	0.875
3 bits	0.002	1.0	1.848	0.726	0.740	0.749	0.816	0.721	$< 10^{-4}$	1.0	2.350	0.737	0.006	0.998	0.886	0.861
3.17 bits	$< 10^{-3}$	1.0	2.093	0.715	0.048	0.984	0.693	0.844	$< 10^{-4}$	1.0	2.373	0.730	0.011	0.996	1.012	0.859
4 bits	$< 10^{-3}$	1.0	2.060	0.721	0.037	0.987	0.739	0.843	$< 10^{-4}$	1.0	2.420	0.727	0.004	0.999	0.988	0.862
4.08 bits	$< 10^{-3}$	1.0	2.123	0.715	0.035	0.989	0.803	0.846	$< 10^{-4}$	1.0	2.559	0.719	0.005	0.998	1.043	0.863

Table 5
Summary of the results for the models with transformer blocks (VIT1, simpler, and VIT2, more complex model).

Resolution	VIT1 (No Aug)				VIT1 (Aug)				VIT2 (No Aug)				VIT2 (Aug)			
	Train		Val		Train		Val		Train		Val		Train		Val	
	Loss	Acc	Loss	Acc	Loss	Acc	Loss	Acc	Loss	Acc	Loss	Acc	Loss	Acc	Loss	Acc
Standard	$< 10^{-3}$	1.0	1.161	0.741	0.223	0.925	0.561	0.812	$< 10^{-3}$	1.0	1.343	0.741	0.099	0.971	0.540	0.835
1 bit	0.032	0.997	1.206	0.646	0.764	0.727	0.883	0.690	0.002	1.0	1.355	0.697	0.331	0.889	0.635	0.783
1.5 bits	0.001	1.0	1.106	0.724	0.559	0.804	0.790	0.723	0.001	1.0	1.430	0.719	0.191	0.937	0.588	0.808
2 bits	0.001	1.0	1.192	0.713	0.473	0.837	0.759	0.735	$< 10^{-3}$	1.0	1.369	0.713	0.358	0.880	0.727	0.750
2.32 bits	$< 10^{-3}$	1.0	1.180	0.729	0.549	0.811	0.772	0.726	$< 10^{-3}$	1.0	1.460	0.724	0.124	0.961	0.559	0.823
3 bits	$< 10^{-3}$	1.0	1.162	0.734	0.624	0.785	0.820	0.709	$< 10^{-3}$	1.0	1.431	0.720	0.113	0.966	0.561	0.822
3.17 bits	$< 10^{-3}$	1.0	1.187	0.722	0.242	0.919	0.582	0.810	$< 10^{-3}$	1.0	1.465	0.727	0.098	0.970	0.563	0.830
4 bits	$< 10^{-3}$	1.0	1.189	0.722	0.238	0.920	0.562	0.814	$< 10^{-3}$	1.0	1.460	0.727	0.109	0.966	0.570	0.823
4.08 bits	$< 10^{-3}$	1.0	1.176	0.738	0.207	0.931	0.551	0.818	$< 10^{-3}$	1.0	1.411	0.725	0.109	0.966	0.580	0.820

Analyzing the memory reduction achieved alongside the comparative performance of low-resolution weight models, we find that the optimal balance between performance and memory requirements is exhibited by the model with 2.32-bit weights ($N_{\text{values}} = 5$). This model

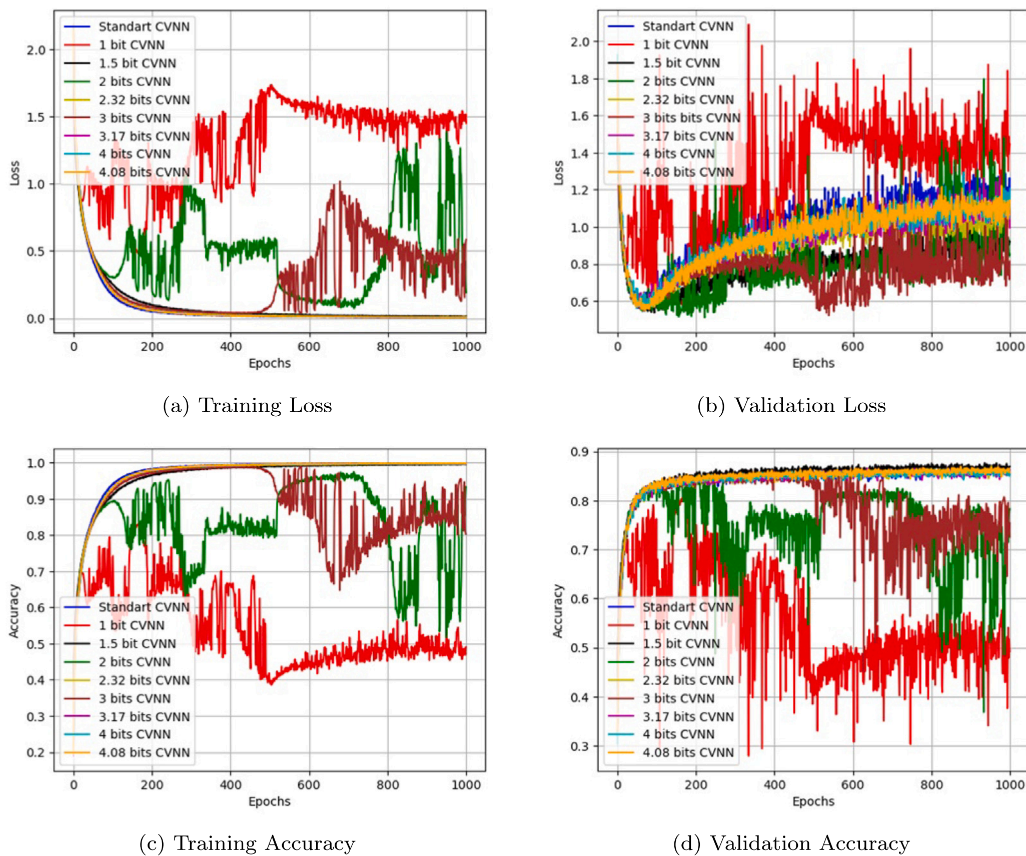


Fig. 9. Training and validation results for the MORE complex convolutional layer models (CVNN2) using data augmentation for various resolutions used in the weights.

Table 6
Reduction in memory usage of low-resolution weight models compared to 32-bit weight models.

Number of bits	Nvalues	Weights stored by byte	Memory reduction
1	2	1	32
1.5	3	5	20
2	4	4	16
2.32	5	3	12
3	8	2	8
3.17	9	2	8
4	16	2	8
4.08	17	1	4

demonstrates a 12-fold reduction in memory usage while maintaining stability during training across all model types, including fully connected layers, convolutional models, and transformer models.

Moreover, while not the primary focus of this work, it is important to acknowledge the well-known challenge of high energy consumption in AI training and the potential benefits of low-resolution weights. The proposed model consumes the same amount of energy as standard neural networks for training because in this stage an auxiliary set of weights with full-precision are used. However, during inference only the low-resolution weights are used and the auxiliary full-precision weights are discard. Thus, during inference the low precision weights not only reduce memory needs but also significantly lower energy consumption due to reduced computational and memory access demands. For example, packing multiple low-resolution parameters into a single byte is straightforward and computationally efficient, ensuring that reducing parameter resolution does not introduce additional latency.

9. Conclusions

In this study, we analyze the numbers of bits required for representing layer weights to achieve performance comparable to 32-bit resolution models. The focus is on multiclass object classification in images, examining models that utilize fully connected layers, convolutional layers, and transformer blocks, with weight resolutions ranging from 1 bit to 4.08 bits. Our approach deliberately avoids employing complex models aimed at maximizing performance or entirely eliminating overfitting.

9.1. About low-resolution weight performances

The primary goal is to determine whether low-resolution weight models can achieve similar performance levels to 32-bit models and generalize learning effectively through comparative analysis. It is important to note that no specialized algorithms were implemented to optimize the use of fewer bits in the weights, which remains an area for future exploration, especially with advancements in computer architectures capable of efficiently handling both small-bit integers and real numbers.

Several key conclusions emerge from this research:

Performance of low-resolution models: Initially, low-resolution models with small number of parameters yield results comparable to standard 32-bit models, although they require more training epochs. Despite this increased training time, low-resolution weight models can potentially be trained more rapidly, as calculations may be executed more efficiently due to the reduced bit representation. However, realizing this advantage requires

Algorithm 6 Forward propagation process of the VIT models.

Require: Input image **x**, patch dimension $patch_size$, number of patches $num_patches$ embedding dimension emb_dim, number of transformer blocks $transformer_layers$, vector with numbers of units in the fully connected layers of the transformer blocks **transformer_units**, number of weight values N_{values}, number of heads num_heads
Ensure: Predicted output **ypred**

```
1:  # Create patches
2:  patches = Patches(patch_size)(x)
3:  # Encode patches
4:  encoded_patches = PatchEncoder(num_patches, emb_dim)(patches)
5:  # Create multiple layers of Transformer Blocks.
6:  for i = 1 to transformer_layers do
7:      # Normalization layer 1
8:      x1 = LayerNormalization(encoded_patches)
9:      # Create a multi-head attention layer.
10:     attention_output, _ = Attention(emb_dim, num_heads, dropout_rate = 0.1)([x1, x1])
11:     # Skip connection 1
12:     x2 = Add(attention_output, encoded_patches)
13:     # Normalization layer 2
14:     x3 = LayerNormalization(x2)
15:     # MLP
16:     x3 = FCL(units = transformer_units[0], N_values, activation = gelu)(x3)
17:     matbfx3 = Dropout(dropout_rate = 0.1)(x3)
18:     x3 = FCL(units = transformer_units[1], N_values, activation = gelu)(x3)
19:     x3 = Dropout(dropout_rate = 0.1)(x3)
20:     # Skip connection 2.
21:     encoded_patches = Add(x3, x2)
22: end for
23: # Create a [batch_size, projection_dim] tensor.
24: representation = LayerNormalization(encoded_patches)
25: representation = Flatten(representation)
26: representation = Dropout(dropout_rate = 0.5)(representation)
27: # MLP
28: features = FCL(units = mlp_head_units[0], N_values, activation = gelu)(representation)
29: features = Dropout(dropout_rate = 0.5)(features)
30: features = FCL(units = mlp_head_units[1], N_values, activation = gelu)(features)
31: features = Dropout(dropout_rate = 0.5)(features)
32: # Classify outputs.
33: ypred = FCL(units = num_classes, N_values, activation = softmax)(features)
```

the development of dedicated hardware capable of performing operations with numbers represented by fewer than 8 bits.

Impact of data augmentation: Data augmentation appears to induce instability in the training of low-resolution weight models, particularly those with a small number of parameters. In contrast, models with a greater number of parameters demonstrate more stable training outcomes with data augmentation, especially those that accommodate zero as a possible weight value.

Preference for odd weight values: A significant finding is that using an odd number of possible weight values in low-resolution models — ensuring the inclusion of zero — yields better performance outcomes. Models with even N_{values}, particularly those utilizing convolutional layers and transformer blocks, tend to exhibit training instability.

Computational optimization: Current computing systems are optimized for a minimum resolution of 8 bits (1 byte), indicating no computational speed advantage when comparing 1-bit to 8-bit resolution weights. However, training models with 8-bit weights can deliver performance comparable to models with 16 or 32-bit weights. It is crucial to note that post-training quantization from 16 or 32 bit weights to 8 bits often leads to performance degradation.

Advantages of low-resolution models: Low-resolution weight models present a significant advantage by enabling the development of more complex models with higher processing units while using substantially less memory compared to 32-bit resolution models or even quantized 8-bit models. These models hold great promise for facilitating the deployment of large language models in embedded devices.

In summary, our findings indicate that using weights with 2.32 bits ($N_{\text{values}} = 5$) strikes the best balance between memory reduction, model performance, and efficiency. However, these conclusions remain preliminary, and further research is needed to explore their generalizability across different architectures, tasks, and deployment scenarios.

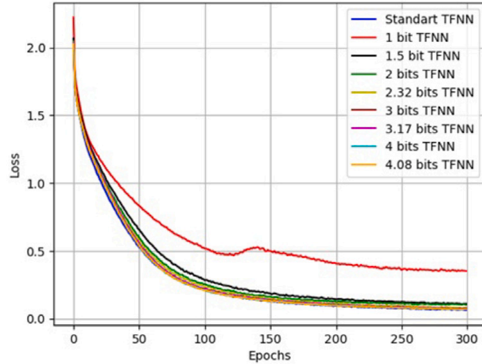
9.2. Future steps

Low-resolution weights have significant potential for improving the efficiency of large-scale models, particularly in terms of memory usage and computational cost. Since our work has demonstrated that 2.32-bit quantization is effective under controlled conditions, the next steps should focus on validating its broader applicability across diverse datasets and architectures, refining the theoretical and practical compression limits, assessing the robustness of training techniques against precision loss, and ultimately exploring its feasibility for real-world deployment in various hardware environments. In details,

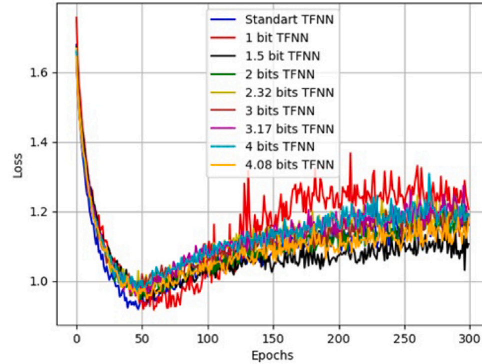
Algorithm 7 Calculation process for Attention Mechanism.

Require: Inputs **x**, embedding dimension emb_dim, number of weight values N_{values}, number of heads num_heads, dropout rate $dropout_rate$
Ensure: Predict output **output**

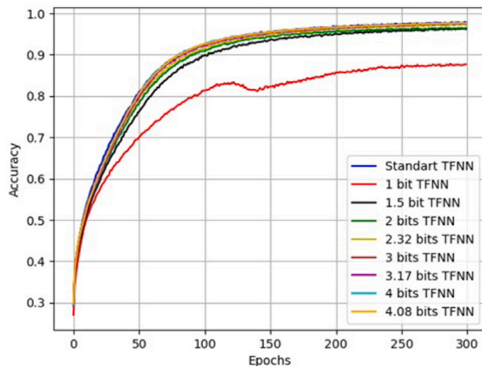
1: # Retrives query, key and value from input list
2: **query**, **key**, **value** ← **x**
3: # Calculate n_key and batch_size values
4: $n_key = \frac{emb_dim}{num_heads}$
5: $batch_size$ = **key**.`shape`[0]
6: # Calculate Q, K and V matrices
7: **Q** = FLC(units = emb_dim, N_{values}, activation = `linear`, use_bias = False)(**query**)
8: **K** = FLC(units = emb_dim, N_{values}, activation = `linear`, use_bias = False)(**key**)
9: **V** = FLC(units = emb_dim, N_{values}, activation = `linear`, use_bias = False)(**value**)
10: # Reshape and permute Q, K and V
11: **Q** = `reshape`(**Q**, [$batch_size$, −1, num_heads, n_key])
12: **Q** = `permute`(**Q**, [0, 2, 1, 3])
13: **K** = `reshape`(**K**, [$batch_size$, −1, num_heads, n_key])
14: **K** = `permute`(**K**, [0, 2, 1, 3])
15: **V** = `reshape`(**V**, [$batch_size$, −1, num_heads, n_key])
16: **V** = `permute`(**V**, [0, 2, 1, 3])
17: # Calculate dot product Q by K
18: **QK** = `matmul`(**Q**, `permute`(**K**, [0, 1, 3, 2]))/$\sqrt{n_key}$
19: # Calculate attention probabilities
20: **attn_prob** = `softmax`(**QK**, axis = −1)
21: # Calculate attention
22: **A** = `matmul`(`dropout`($dropout_rate$)(**attn_prob**), **V**)
23: **A** = `permute`(**A**, [0, 2, 1, 3])
24: **A** = `reshape`(**A**, [$batch_size$, −1, $num_heads \cdot n_key$])
25: # Calculate output using FCL
26: **output** = FLC(units = emb_dim, N_{values}, activation = `linear`, use_bias = False)(**A**)



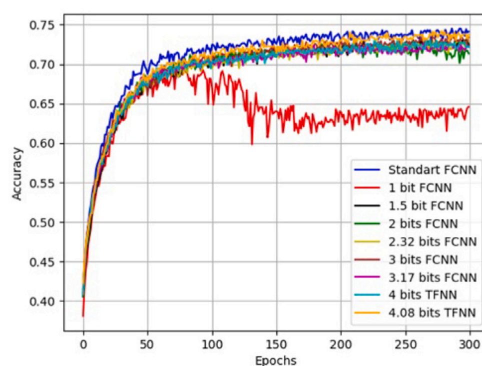
(a) Training Loss



(b) Validation Loss



(c) Training Accuracy



(d) Validation Accuracy

Fig. 10. Training and validation results for the simpler VIT models (VIT1) for various resolutions used in the weights.

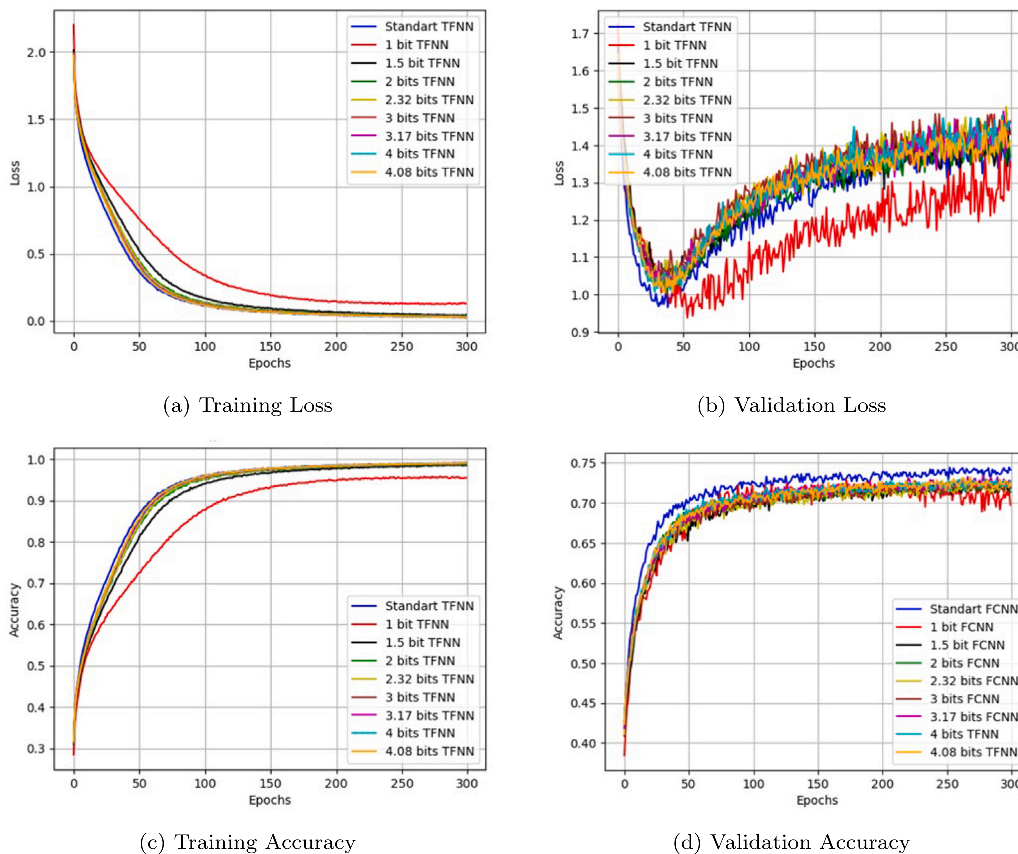


Fig. 11. Training and validation results for more complex VIT models (VIT2) for various resolutions used in the weights.

Scalability to larger datasets and architectures: extend the evaluation to larger and more diverse datasets — such as those for language, time series, and image generation — and various neural network architectures, including deeper transformers, convolutional networks, and recurrent models. This will help confirm the generalizability of the approach across different domains and model types;

Compression limits: investigate the theoretical and practical boundaries of quantization by exploring progressively lower resolutions, down to 1 bit. Analyze the trade-offs between model accuracy, stability, and efficiency, identify critical thresholds for performance degradation, and explore mitigation strategies such as adaptive precision scaling, Chen et al. (2024), or entropy-based encoding, Park et al. (2017);

Impact on downstream tasks: assess the effectiveness of quantization across various application domains, including natural language processing, computer vision, and reinforcement learning, to determine whether the benefits extend to diverse use cases;

Hardware Compatibility and Deployment: evaluate the performance of quantized models on real-world hardware platforms, such as GPUs, TPUs, and edge devices, to understand their behavior under practical constraints and optimize deployment strategies.;

Energy efficiency and latency analysis: conduct detailed profiling of power consumption and inference latency to quantify the practical efficiency gains of using low-resolution weights.

CRediT authorship contribution statement

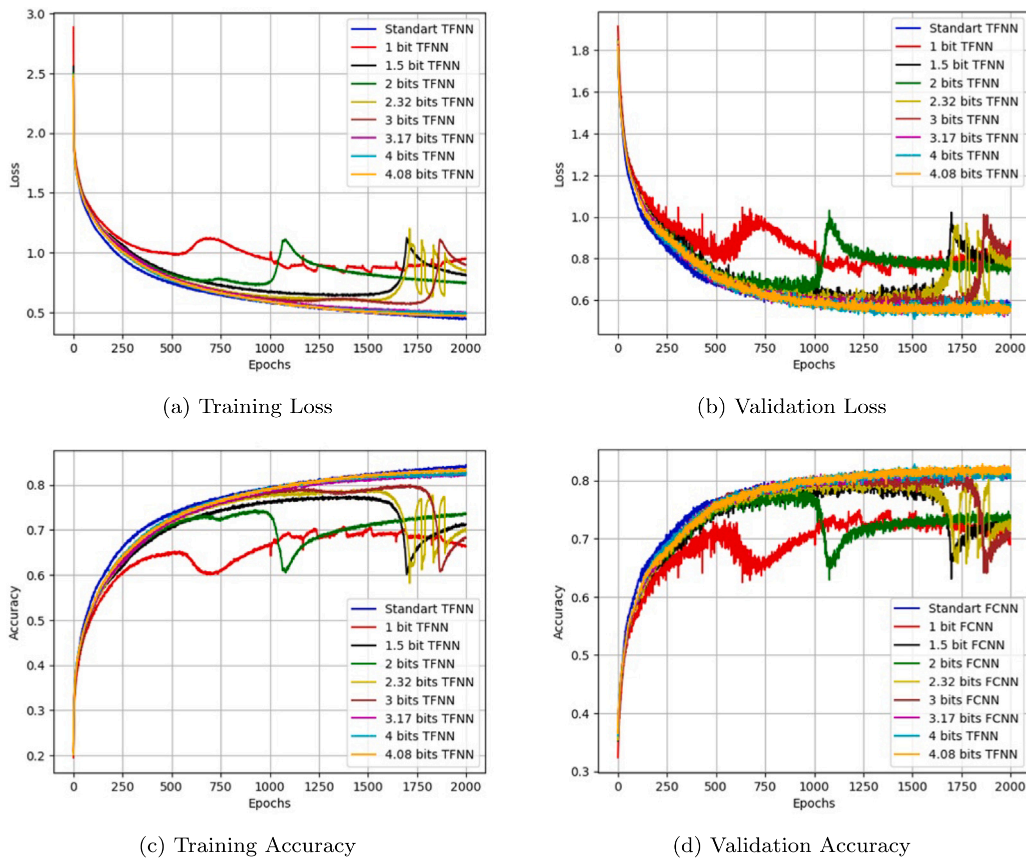
Eduardo Lobo Lustosa Cabral: Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Project administration, Methodology, Formal analysis, Data curation, Conceptualization. **Larissa Driemeier:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Methodology, Investigation, Data curation, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. Python class for fully connected layers

```
1  import tensorflow as tf
2  from tensorflow.keras.layers import Layer
3
4  # Define a custom dense layer with activation inhibition
5  class DenseBit(Layer):
6
7      # Method for initializing the class
8      def __init__(self, units=32, nvalues=2, activation=None, **kwargs):
9          super(DenseBit, self).__init__(**kwargs)
10
11         # Initialize the number of units
12         self.units = units
13
14         # Initialize the number of quantization levels
15         self.nvalues = nvalues
16
17         # Define the activation function
18         self.activation = tf.keras.activations.get(activation)
19
20     # Method executed when an object of the class is instantiated
21     def build(self, input_shape):
22         # Initialize kernel as a trainable parameter
23         w_init = tf.keras.initializers.GlorotNormal()
24         self.w = tf.Variable(initial_value=w_init(shape=
25                  (input_shape[-1], self.units),
26                 dtype='float32'), trainable=True)
27
28         # Initialize biases
29         b_init = tf.zeros_initializer()
30         self.b = tf.Variable(initial_value=
31                  b_init(shape=(self.units,),
```



(a) Training Loss (b) Validation Loss

(c) Training Accuracy (d) Validation Accuracy

Fig. 12. Training and validation results for the simpler VIT models (VIT1) using data augmentation for various resolutions used in the weights.

```
32                dtype='float32'), trainable=True)
33
34      def quantD(self, x):
35          # Perform quantization
36          vmax = (tf.cast(self.nvalues, tf.float32) - 1.0) / 2.0
37          q = tf.math.round(vmax * x + vmax)
38          q = (q - vmax) / vmax
39          q = tf.where(q > 1.0, 1.0, q)
40          q = tf.where(q < -1.0, -1.0, q)
41          return q
42
43      def get_config(self):
44          config = super().get_config()
45          config.update({"units": self.units, "nvalues": self.nvalues})
46          return config
47
48      # Define computations executed by the layer using the selected activation
        function
49      def call(self, inputs):
50          # Compute adjustment factor
51          w_mean = tf.reduce_mean(tf.math.abs(self.w))
52          self.factor = 2.0 * w_mean + 1.0e-06
53
54          # Quantize weights
55          w_adjusted = self.w / self.factor
56
57          if self.trainable:
58              wq = w_adjusted + tf.stop_gradient(self.quantD(w_adjusted) -
            w_adjusted)
59          else:
60              wq = self.quantD(w_adjusted)
61
62          # Compute activations
63          activations = self.activation(tf.matmul(self.factor * inputs, wq) +
            self.b)
64          return activations
```

Appendix B. Python class for convolutional layers

```
1  import tensorflow as tf
2  from tensorflow.keras.layers import Layer
3
4  # Define a custom dense layer with activation inhibition
5  class Conv2DBit(Layer):
6
7      # Method for initializing the class
8      def __init__(self, units, kernel_size, nvalues, padding='SAME',
9                   activation=None, **kwargs):
10
11         super(Conv2DBit, self).__init__(**kwargs)
12
13         # Initialize the number of units
14         self.units = units
15
16         # Filter dimensions
17         self.kernel_size = kernel_size
18
19         # Initialize the number of quantization levels
20         self.nvalues = nvalues
21
22         # Define the activation function
23         self.activation = tf.keras.activations.get(activation)
24
25         # Define padding
26         self.padding = padding
27
28     def build(self, input_shape):
29         # Initialize kernel as a trainable parameter
30         self.w = self.add_weight(name='kernel',
31                                  shape=(self.kernel_size[0], self.kernel_size
       [1],
32                                         input_shape[-1], self.units),
33                                  initializer='glorot_uniform',
34                                  dtype='float32',
35                                  trainable=True)
36
37         # Initialize biases
38         self.b = self.add_weight(name='bias',
39                                  shape=(self.units,),
40                                  initializer='zeros',
41                                  dtype='float32',
42                                  trainable=True)
43
44     def quantD(self, x):
45         # Perform quantization
46         vmax = (tf.cast(self.nvalues, tf.float32) - 1.0) / 2.0
47         q = tf.math.round(vmax * x + vmax)
48         q = (q - vmax) / vmax
49         q = tf.where(q > 1.0, 1.0, q)
50         q = tf.where(q < -1.0, -1.0, q)
51         return q
52
53     def get_config(self):
54         config = super().get_config()
55         config.update({"units": self.units, "kernel_size": self.kernel_size, "
       nvalues": self.nvalues, 'padding': self.padding})
56         return config
57
58     # Define computations executed by the layer using the selected activation
       function
59     def call(self, inputs):
60         # Compute adjustment factor
61         w_mean = tf.reduce_mean(tf.math.abs(self.w))
62         self.factor = 2.0 * w_mean + 1.0e-06
```

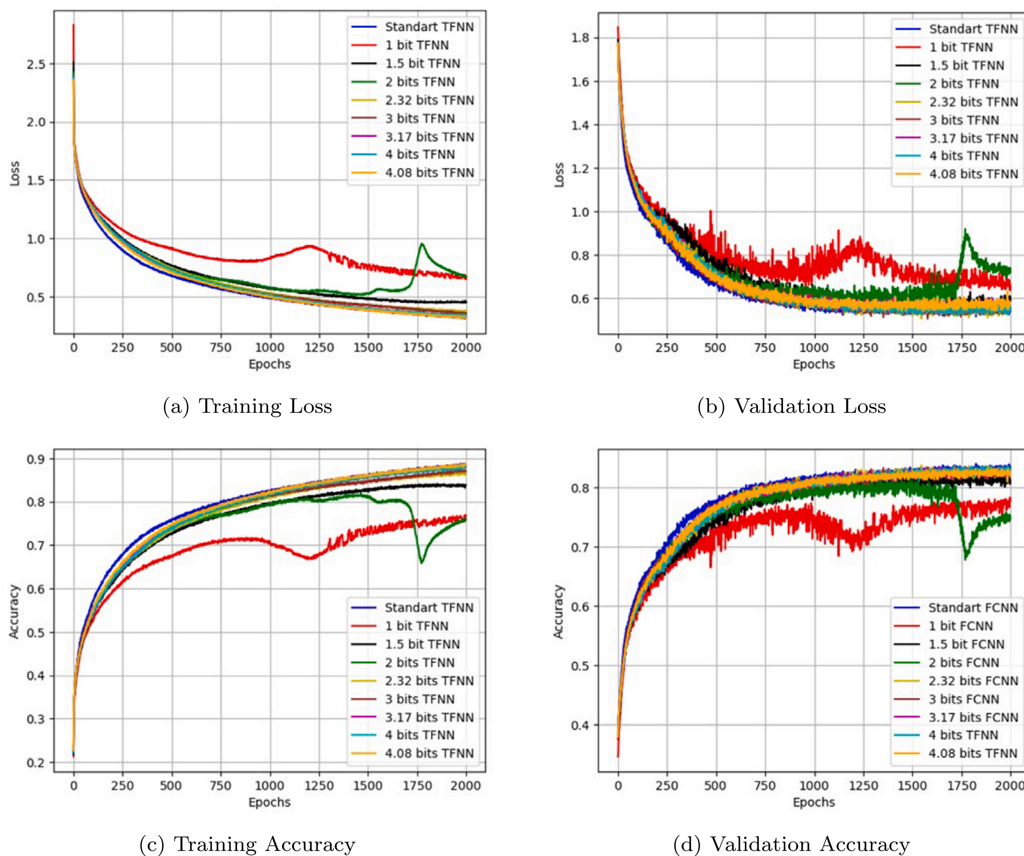


Fig. 13. Training and validation results for more complex VIT models (VIT2) using data augmentation for various resolutions used in the weights.

```
63
64         # Quantize weights
65         w_adjusted = self.w / self.factor
66
67         if self.trainable:
68             wq = w_adjusted + tf.stop_gradient(self.quantD(w_adjusted) −
       w_adjusted)
69         else:
70             wq = self.quantD(w_adjusted)
71
72         # Compute activations
73         activations = self.activation(tf.nn.conv2d(self.factor ∗ inputs, wq,
74                                       strides=(1, 1), padding=self.padding) +
       self.b)
75
76         return activations
```

Appendix C. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.neunet.2025.107763.

Data availability

Data will be made available on request.

References

Alcorn, M. A. (2023). A minimal PyTorch implementation of the VQ-VAE model described in “neural discrete representation learning”. https://github.com/airalcorn2/vqvae-pytorch?tab=readme-ov-file.

Chen, J., Yang, Q., Tian, S., & Zhang, S. (2024). Adaptive quantization with mixed-precision based on low-cost proxy. In *ICASSP 2024 - 2024 IEEE international conference on acoustics, speech and signal processing* (pp. 6720–6724). IEEE, http://dx.doi.org/10.1109/icassp48485.2024.10447866.

Chu, T., Luo, Q., Yang, J., & Huang, X. (2021). Mixed-precision quantized neural networks with progressively decreasing bitwidth. *Pattern Recognition*, *111*, Article 107647. http://dx.doi.org/10.1016/j.patcog.2020.107647.

Courbariaux, M., Bengio, Y., & David, J.-P. (2015). BinaryConnect: training deep neural networks with binary weights during propagations. In *Proceedings of the 28th international conference on neural information processing systems: Vol. 2,* (pp. 3123–3131). Cambridge, MA, USA: MIT Press.

Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R., & Bengio, Y. (2016). Binarized neural networks: Training neural networks with weights and activations constrained to +1 or 1. arXiv preprint arXiv:1602.02830v3.

Dai, H., Wu, J., Wang, Y., Yen, J., Zhang, Y., & Xu, C. (2023). Cost-efficient sharing algorithms for DNN model serving in mobile edge networks. *IEEE Transactions on Services Computing*, *16*(4), 2517–2531. http://dx.doi.org/10.1109/TSC.2023.3247049.

de Lima, J. P. C., & Carro, L. (2022). Quantization-aware in-situ training for reliable and accurate edge AI. In *2022 design, automation & test in Europe conference & exhibition*. http://dx.doi.org/10.23919/DATE54114.2022.9774657.

Deng, L., Jiao, P., Pei, J., Wu, Z., & Li, G. (2018). GXNOR-net: Training deep neural networks with ternary weights and activations without full-precision memory under a unified discretization framework. *Neural Networks*, *100*, 49–58. http://dx.doi.org/10.1016/j.neunet.2018.01.010.

Gong, R., Yong, Y., Gu, S., Huang, Y., Zhang, Y., Liu, X., & Tao, D. (2024). LLM-qbench: A benchmark towards the best practice for post-training quantization of large language models. arXiv preprint arXiv:2405.06001v1.

Gupta, S., Agrawal, A., Gopalakrishnan, K., & Narayanan, P. (2015). Deep learning with limited numerical precision. In *International conference on machine learning.*

Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., & Bengio, Y. (2022). Quantization-aware in-situ training for reliable and accurate edge AI. In *NeurIPS 2016, Proceedings of the 30th conference on neural information processing systems.*

Ignatov, D., & Ignatov, A. (2020). Controlling information capacity of binary neural network. *Pattern Recognition Letters*, *138*, 276–281. http://dx.doi.org/10.1016/j.patrec.2020.07.033.

Kirtas, M., Oikonomou, A., Passalis, N., Mourgias-Alexandris, G., Moralis-Pegios, M., Pleros, N., & Tefas, A. (2022). Quantization-aware training for low precision photonic neural networks. *Neural Networks*, *155*, 561–573. http://dx.doi.org/10.1016/j.neunet.2022.09.015.

Krizhevsky, A. (2009). *Learning multiple layers of features from tiny images: Technical report*, Computer Science Department, University of Toronto.

Liu, Z., Wang, Y., Han, K., Zhang, W., Ma1, S., & Gao, W. (2021). Post-training quantization for vision transformer. In *NeurIPS, Proceedings of the 35th conference on neural information processing systems.*

Ma, S., Wang, H., Ma, L., Wang, L., Wang, W., Huang, S., Dong, L., Wang, R., Xue, J., & Wei, F. (2024). The era of 1-bit LLMs: All large language models are in 1.58 bits. arXiv preprint arXiv:2402.17764v1.

Moosmann, J., Müler, H., Zimmerman, N., Rutishauser, G., Benini, L., & Magno, M. (2024). Flexible and fully quantized lightweight TinyissimoYOLO for ultra-low-power edge systems. *IEEE Access*, *12*, 75093–75107. http://dx.doi.org/10.1109/ACCESS.2024.3404878.

Park, E., Ahn, J., & Yoo, S. (2017). Weighted-entropy-based quantization for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*.

Peng, H., & Chen, S. (2019). BDNN: Binary convolution neural networks for fast object detection. *Pattern Recognition Letters*, *125*, 91–97. http://dx.doi.org/10.1016/j.patrec.2019.03.026.

Qin, H., Gong, R., Liu, X., Bai, X., Song, J., & Sebe, N. (2020). Binary neural networks: A survey. *Pattern Recognition*, *105*, Article 107281. http://dx.doi.org/10.1016/j.patcog.2020.107281.

Rastegari, M., Ordonez, V., Redmon, J., & Farhadi, A. (2016). XNOR-net: ImageNet classification using binary convolutional neural networks. CoRR abs/1603.05279. arXiv:1603.05279.

Siddegowda, S., Fournarakis, M., Nagel, M., Blankevoort, T., Patel, C., & Khobare, A. (2022). Neural network quantization with AI model efficiency toolkit (AIMET). arXiv preprint arXiv:2201.08442v1.

Tang, Z., Peng, X., Li, K., & Metaxas, D. N. (2020). Towards efficient U-nets: A coupled and quantized approach. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *42*(8), 2038–2050. http://dx.doi.org/10.1109/TPAMI.2019.2907634.

van den Oord, A., Vinyals, O., & Kavukcuoglu, K. (2017). Neural discrete representation learning. CoRR abs/1711.00937. arXiv:1711.00937.

Wang, H., Ma, S., Dong, L., Huang, S., Huaijie Wang, L. M., Yang, F., Wang, R., Wu, Y., & Wei, F. (2023). BitNet: Scaling 1-bit transformers for large language models. arXiv preprint arXiv:2310.11453v1.

Yang, Y., Deng, L., Wu, S., Yan, T., Xie, Y., & Li, G. (2020). Training high-performance and large-scale deep neural networks with full 8-bit integers. *Neural Networks*, *125*, 70–82. http://dx.doi.org/10.1016/j.neunet.2019.12.027.