# Post training optimisation of CNN for image classification on resource constrained devices

|  |  |  |
| --- | --- | --- |
| Zulfiqar Mohammadi  student  a1850245@adelaide.edu.au |  | Dr. Xinyu Wang  Supervisor  xinyu.wang@adelaide.edu.au |

ABSTRACT

Deep learning models, particularly Convolutional Neural Networks (CNNs), have achieved remarkable success in computer vision tasks such as image classification. However, their high computational and memory demands make it difficult to deploy on resource-constrained devices like mobile phones, embedded systems, and microcontrollers. This research investigates post-training optimisation techniques, with a focus on quantization, to address this limitation. Unlike traditional methods that require retraining, post-training quantization reduces model size and inference time by converting high-precision parameters to lower-precision formats (e.g., 32-bit floats to 8-bit integers), all while preserving model accuracy. This research evaluates the impact of quantization on popular lightweight CNN models, such as MobileNet, and demonstrates how these models can be effectively deployed in real-world constrained environments. The results offer practical insights and performance trade-offs, contributing to the broader goal of enabling efficient deep learning on edge devices.

CCS CONCEPTS

Convolutional Neural Networks (CNNs), Post-Training Optimisation, Quantization, resource-constrained environments (RCEs)

1 INTRODUCTION AND MOTIVATION

Deep learning has made significant progress in solving complex computer vision tasks such as image classification. Convolutional Neural Networks (CNNs), in particular, have shown remarkable accuracy in such tasks. However, achieving high performance comes at the cost of increased model complexity and size. These models typically require powerful GPUs for training and inference, which makes them unsuitable for deployment on devices with limited computational resources, such as mobile phones and embedded systems.

In the past although mobile-friendly CNN architectures have been proposed to address this issue, they often compromise accuracy in favor of speed and memory efficiency. This trade-off limits their usability in real-world applications where both accuracy and efficiency are important. Moreover, many existing optimisation techniques require retraining the model from scratch, which is time-consuming and computationally expensive. This poses a major challenge for developers who need to deploy accurate and efficient models on edge devices with restricted resources.

The core problem this research aims to address is how to reduce the size and computation requirements of CNN models while maintaining their accuracy, without retraining. This is where post-training optimisation techniques, particularly quantization, offer a practical solution. Quantization allows us to convert high-precision (e.g., 32-bit float) model weights and activations into lower-precision formats (e.g., 8-bit integers), leading to reduced model size, lower energy consumption, and faster inference, all with minimal loss in accuracy.

My motivation for pursuing this research comes from a deep interest in expanding my knowledge in Machine Learning concepts and developing efficient AI systems that can function reliably in resource-constrained environments (RCEs), such as those found in mobile healthcare and embedded vision.

This study explores the use of post-training quantization techniques to optimise CNN models for image classification tasks on edge devices. The methodology involves selecting suitable CNN architectures, applying quantization methods without retraining, and evaluating the impact on model size, speed, and accuracy. Popular models such as MobileNet are used as benchmarks due to their balance of accuracy and efficiency. All experiments are conducted under constraints similar to those found in real-world embedded systems to ensure practical relevance.

This research aims to contribute to addressing the above discussed challenge. For example, systematically evaluating how post-training quantization affects model accuracy, size, and inference time on resource-constrained devices. Additionally, comparing the performance of quantized MobileNet and other lightweight CNNs to identify optimal trade-offs for deployment. Lastly, providing practical guidelines and performance metrics to aid students or users in choosing quantization strategies for real-world applications. With these contributions, I aim to bridge the gap between high-performing deep learning models and the limitations of edge devices.

2 LITERATURE REVIEW

This section reviews key research and techniques aimed at optimizing DNNs for deployment on edge devices.

2.1 Efficient Acceleration of Deep Learning Inference

2.1.1 Scope and Contributions

Early studies on deploying DNNs for edge computing aimed to eliminate cloud dependency. This was done to make systems faster, protect user privacy, and reduce the amount of data sent over the internet. These works introduced compression strategies such as quantization, pruning, and hardware-aware architectures to address memory and energy constraints. For further compression strategies and details, refer to the scientific paper mentioned in 2.1.

However, these approaches often required retraining or redesigning models, limiting their accessibility and practicality. For example, retraining-based pruning or distillation techniques assume access to full datasets and training infrastructure. This assumption does not hold in many real-world scenarios.

**Connection to this research:** This research takes a different approach by focusing only on post-training quantization (PTQ), which lets us make models smaller and faster without needing to retrain them or change their structure. This makes it more feasible for on-device optimization, particularly when pre-trained models are available.

2.1.2 Quantization selection and fundamentals

Quantization was chosen as the compression technique because it plays a vital role in enabling neural networks to operate efficiently on devices with limited computing power and memory. It strikes a good balance between maintaining accuracy and improving speed and resource use. This makes it ideal for deploying AI models in real-world, everyday applications where high performance and low power usage are both important. Quantization techniques reduce the bit-width of model parameters and activations, typically from FP32 to INT8 or FP16.

While this significantly reduces computational and memory demands, it comes with trade-offs in accuracy and numerical stability. For example, Table 3 (page 60) demonstrates up to 49× improvement in size reduction when used in conjunction with other optimization strategies. However, the overall performance degrades in model with high dynamic range layers or sensitive activation functions. This could be due to the loss of information caused by reducing bit width, which limits the precision with which weights and activations are stored and processed.

2.1.3 Comparing Quantization Approaches

Prior research categories quantization into linear vs. nonlinear and uniform vs. mixed precision strategies. Linear quantization is widely supported in hardware and easily scalable but often fails when distributions are highly skewed. Nonlinear quantization (e.g., using logarithmic encoding) can better approximate outliers but introduces decoding overhead and implementation complexity. This research selects linear quantization with strategic precision-mixing because it aligns with current edge hardware capabilities.

2.1.4 Challenges and Unresolved Issues

Despite numerous advancements, quantization remains liable to accuracy degradation in deep or highly non-linear models, especially in convolution-heavy architectures like ResNet. Furthermore, the challenge of selecting appropriate quantization techniques for certain pre-trained CNN architectures is still not declared.

**Motivation for our work:** This research directly addresses this issue by evaluating quantization sensitivity per architecture. For example, measuring accuracy, inference and model size by applying quantization techniques on light weight deep learning models. To address the challenge of maintaining accuracy while reducing size and energy consumption, CIFAR-100 will be utilized for training these deep learning models. This dataset reveals performance drop-offs more clearly due to its fine-grained class distinctions.

2.2 A Method of Deep Learning Model Optimisation

2.2.1 Project scope

This study evaluates the effectiveness of various quantization methods including Dynamic range quantization, Full Integer Quantization, and FP16 Quantization, on traditional and lightweight CNN architectures (e.g., ResNet, EfficientNetB0, MobileNet). The models are tested on the CIFAR-100 dataset, assessing inference speed and accuracy trade-offs.

2.2.2 Dynamic range quantization (DRQ)

This is the simplest form of post-training quantization which statically quantizes the weights from floating point to 8-bits of precision and dynamically quantizes the activations at inference. This means that the activations are always stored in float 32, however, they are converted to 8-bit integers while processing and back to floating point after the processing is done.

2.2.3 Full Integer Quantization (FIQ)

FIQ quantizes all elements (weights, activations, biases) to INT8. It requires calibration data but allows end-to-end integer arithmetic, which is highly compatible with hardware accelerators like GPUs/TPUs. While highly efficient, it may slightly reduce model accuracy depending on calibration quality.

2.2.4 FP16 Quantization

FP16 reduces precision while maintaining the floating-point structure. It offers easy implementation, good accuracy retention, and compatibility with modern GPUs. However, its compression is less aggressive (2× vs. 4× for INT8), and it lacks benefits on legacy hardware.

* + 1. Summary of Findings

Each method presents a trade-off between model size, inference speed, and accuracy. Firstly, FIQ offers the best acceleration and compression but needs calibration. Additionally, FP16 is easiest to implement and retains accuracy well. Finally, BLQ provides moderate improvements but adds some runtime costs. While all three approaches have merit, none fully solves the challenge of ultra-efficient deployment. This supports the need for further optimization and hybrid strategies in an area our research investigates.

2.3 Summary/comparison and Critical Insights

Across all reviewed works, quantization emerges as a consistently effective and lightweight method for reducing DNN resource requirements. However, there is no one-size-fits-all solution, and trade-offs between model size, speed, and accuracy are evident in every technique. By targeting the critical gap between high-performance CNNs and low-power deployment needs, the aim is to push forward the practical usability of DNNs in edge computing scenarios

Our research builds upon these findings by firstly, focusing exclusively on post-training quantization, avoiding the cost of retraining. Secondly, experimenting with hybrid quantization strategies (e.g., INT8/FP16) for balancing accuracy and performance. Lastly, evaluating these methods on practical image classification tasks using CIFAR-100 and widely used models like MobileNet and ResNet.

3 METHODOLOGY

In this project, we used the CIFAR-100 dataset to evaluate and compare the performance of deep learning models for image classification. CIFAR-100 is a popular benchmark dataset that contains 60,000 colour images, each sized 32 by 32 pixels. These images are divided into 100 different categories, with 50,000 images used for training and 10,000 for testing. The goal was to train models that could correctly classify new images into one of the 100 categories.

Since many of the advanced models we used require input images to be larger than 32×32, we resized all images to 224×224 pixels. We also applied a special preprocessing step based on the specific model being used, to make sure the input matched what the model was trained on originally. The labels (or class names) were converted into one-hot encoded format so that the model could output probabilities for all 100 classes.

We worked with several popular deep learning models such as ResNet50, EfficientNetB0, and MobileNetV2. These models were not trained from scratch. Instead, we used versions that were already trained on a large dataset called ImageNet. We used them as feature extractors and added a few new layers on top to adapt them to the CIFAR-100 classification task. These new layers included a global average pooling layer, a dense (fully connected) layer with 500 neurons and ReLU activation, a dropout layer to prevent overfitting, and finally a softmax output layer with 100 neurons (one for each class).

Because of limited computing resources, we could not train all the models at once. Instead, we trained each model one by one and saved its results in a separate file. Once all models were trained and tested, the results were collected and combined manually into a final file for comparison. Each model was trained for 5 epochs using a batch size of 300. We used the Adam optimizer with a small learning rate of 0.0005, and the models were evaluated using accuracy and loss on both the training and test datasets.

In addition to measuring model accuracy and training time, we also evaluated how fast the model could make predictions (inference time), how large the saved model file was (model size), and how many parameters it contained. To further explore model efficiency, we applied three types of quantization to each trained model using TensorFlow Lite: dynamic range quantization, float16 quantization, and integer (INT8) quantization. These methods help reduce the size of the model and speed up prediction, especially on devices with limited resources like mobile phones or embedded systems.

To measure the performance of each quantized model, we ran predictions on the test data using the TensorFlow Lite interpreter. We recorded how accurate the model was, how long it took to make each prediction, and how small the quantized model file became. We used graphs to compare the accuracy, speed, and size of the models before and after quantization.

All experiments were done using Python and libraries such as TensorFlow, Keras, NumPy, and Matplotlib. The work was run either on Google Colab or on a local machine depending on the availability of resources. This setup allowed us to study the balance between accuracy and efficiency across different models and quantization techniques.

4 EXPERIMENTAL SETUP

In this research, several experiments were performed on image classification. During the experiments, we used multiple architectures from Keras API. All the models’ details such as input shape and preprocessing function were stored in a dictionary to automate the training process.

The models were evaluated on CIFAR-100. This dataset is just like CIFAR-10, except it has 100 classes containing 600 images each. There are 500 training images and 100 testing images per class. The 100 classes in the CIFAR-100 are grouped into 20 super classes.

Image classification evaluation was conducted using multiple performance indicators across original and quantized models. Metrics include train/test accuracy and loss, number of parameters, training and inference time per sample, and model size. Quantization results are reported for Dynamic Range, Float16, and INT8 methods, highlighting trade-offs in accuracy, inference latency (ms), and compression efficiency (MB). This provides a benchmark to assess the impact of each quantization technique on both traditional and lightweight CNN architectures.

Due to hardware resource limitations, it was not feasible to train all models simultaneously. As a result, each model was trained individually, and its results were saved in a separate CSV file. Once all training sessions were completed, the results were manually consolidated into a single CSV file containing performance metrics for each model. This approach ensured organized data collection and consistency across experiments despite sequential execution.

Time per inference step is the average of 300 batches and 5 epochs.

* CPU: AMD Ryzen 7 7730U
* RAM: 16 GB
* GPU: Google Collab T4 (Free Version)
* Batch size: 300

Based on analysis of model size, inference time, and accuracy metrics from the table of available Keras applications (also attached in the appendix), the three most suitable models for edge device deployment are MobileNet, MobileNetV2, and EfficientNetB0. Considering constraints in computational resources and submission deadline, the quantization evaluation and discussion will be limited to these three models.

5 RESULTS

The evaluation of the three convolutional neural network models i.e. MobileNet, MobileNetV2, and EfficientNetB0, reveals clear differences in performance, model complexity, and computational efficiency. Figure 1 indicates that EfficientNetB0 demonstrates the best overall performance, achieving the highest test accuracy of 72% and the lowest test loss of 0.94, indicating strong generalization on unseen data. In comparison, MobileNet and MobileNetV2 reach test accuracies of 66% and 63%, respectively, with higher losses, making them less reliable in prediction tasks.

From a model complexity perspective, MobileNetV2 is the most lightweight, with only 2.94 million parameters and a model size of 16.87 MB, making it suitable for devices with limited memory. MobileNet is slightly larger at 3.80 million parameters and 18.95 MB, while EfficientNetB0, though performing the best, is the most complex with 4.7 million parameters and a size of 23.81 MB.

When considering computational cost, MobileNetV2 offers the fastest training time at 334.69 seconds, followed by MobileNet at 372.53 seconds, whereas EfficientNetB0 takes the longest at 423.58 seconds. However, in terms of inference speed—crucial for deployment in real-time systems—EfficientNetB0 is vastly superior, achieving a remarkably low inference time of just 0.2052 seconds per sample, making it over 20 times faster than MobileNet and more than 37 times faster than MobileNetV2.

From the above discussion and Figure 1, EfficientNetB0 is the best choice for scenarios where accuracy and inference speed are critical, despite its larger size and longer training time. MobileNetV2, on the other hand, is ideal for memory-constrained environments, offering the smallest model footprint with moderate performance. MobileNet strikes a balance between the two, offering moderate accuracy, training time, and model size.

|  |
| --- |
|  |
| Figure 1: Experiment results before quantization |

The graphs shown in Figures 2 to 5 illustrate the training and validation performance of two deep learning models, MobileNet and MobileNetV2, over five epochs. Due to limited computation resources, and submission schedule, the number of epochs had to be reduced to 5 epochs and retraining the EfficientNetB0 was not possible. However, from Figure 1, It is clear that MobileNet and MobileNetV2 are more suitable options for implementation on edge devices compared to EfficientNetB0. Therefore, the training and validation graphs of EfficientNetB0 are not plotted in this section.

In the training accuracy graph (shown in Figure 2), both models show a clear upward trend, indicating that they are effectively learning from the training data. Initially starting with an accuracy of below 40%, both MobileNet and MobileNetV2 demonstrate significant improvements by the end of the fifth epoch. However, MobileNet consistently achieves slightly higher accuracy than MobileNetV2 at each epoch, suggesting that it is learning slightly more efficiently during training.

|  |
| --- |
|  |
| **Figure 2: MobileNet and MobileNetV2 Training Accuracy** |

Looking at the validation accuracy graph (shown in Figure 3), a similar pattern was observed. Both models improve their performance on the unseen testing data over time. MobileNet continues to maintain a lead over MobileNetV2 in terms of validation accuracy across all epochs. By the final epoch, MobileNet reaches an accuracy of around 68%, whereas MobileNetV2 lags slightly behind at just under 65%. This indicates that MobileNet not only learns faster but also generalizes better to new data in this early phase of training.

|  |
| --- |
|  |
| **Figure 3: MobileNet and MobileNetV2 Validation Accuracy** |

The training loss graph (shown in Figure 4) shows that both models show a sharp decrease in training loss from the first to the second epoch, followed by a steady decline in subsequent epochs. MobileNet again maintains a slight edge with consistently lower loss values than MobileNetV2. This lower loss reflects more confident predictions on the training set, implying better model optimization.

|  |
| --- |
|  |
| **Figure 4: MobileNet and MobileNetV2 Training Loss** |

Finally, in the validation loss graph (shown in Figure 5), both models demonstrate decreasing loss over epochs, which is a positive indicator of generalization. MobileNet shows slightly lower validation loss than MobileNetV2 throughout training, reinforcing its superior performance on unseen data. By the fourth epoch, both models reach validation loss values close to 1.1–1.2, but MobileNet remains the better performer.

|  |
| --- |
|  |
| **Figure 5: MobileNet and MobileNetV2 Validation Loss** |

While both MobileNet and MobileNetV2 exhibit good learning behavior with improvements in accuracy and loss, MobileNet shows slightly better performance across all metrics. It not only trains more effectively but also generalize more reliably on the validation set. Given these early training results, MobileNet appears to be the more suitable model for this task, though further training and tuning may alter this outcome.

# Post-training Quantization

After training the models, they were quantized, which is the main goal of this research. The results of this process are shown in Figures 6 to 8 and Tables 1 to 3.

As shown in Figure 6 and Table 1, for MobileNet, the original accuracy of 66.92% dropped slightly to 65.33% with dynamic range quantization, while Float16 quantization slightly improved accuracy to 66.99%, indicating strong compatibility with half-precision operations. However, INT8 quantization caused a notable decline in performance, reducing accuracy to 55.7%, suggesting that full integer quantization compromises the model’s ability to preserve learned features. Similarly, MobileNetV2 experienced only a slight decrease in accuracy with dynamic range (63.43%) and Float16 (63.89%) compared to its original 63.96%, while INT8 quantization led to a moderate reduction to 56.23%. Among all models, EfficientNetB0 had the highest original accuracy of 72.15% and demonstrated exceptional resilience with dynamic range (71.58%) and Float16 quantization (72.23%). However, INT8 quantization significantly degraded its accuracy to just 36.29%, highlighting the model’s sensitivity to aggressive quantization due to its more complex architecture.

Overall, Float16 quantization preserved model performance best across all models, followed closely by dynamic range quantization, while INT8 quantization often resulted in substantial accuracy loss, especially in more complex networks like EfficientNetB0.

|  |
| --- |
|  |
| **Figure 6: Quantization Accuracy Measures** |
|  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Original Accuracy (%) | Dynamic Range Accuracy (%) | Float16 Accuracy (%) | INT8 Accuracy (%) |
| MobileNet | 66.92 | 65.33 | 66.99 | 55.7 |
| MobileNetV2 | 63.96 | 63.43 | 63.89 | 56.23 |
| EfficientNetB0 | 72.15 | 71.58 | 72.23 | 36.29 |
| Table 1: Quantization Accuracy Measures | | | | |

The inference time analysis of the quantized models shown in Figure 7 and Table 2, demonstrates the computational efficiency gained through quantization strategies. MobileNet shows a clear improvement in inference speed with quantization. Its original dynamic range inference time of 25.36 ms is reduced to 19.13 ms with Float16 quantization and 21.27 ms with INT8 quantization. This suggests that both Float16 and INT8 improve runtime performance, with Float16 offering the fastest inference time for MobileNet.

For MobileNetV2, the benefits of quantization are even more pronounced. The dynamic inference time of 20.2 ms is significantly reduced to just 11.46 ms using Float16 quantization, a nearly 43% improvement. INT8 quantization also improves the inference time, bringing it down to 15.82 ms. This makes MobileNetV2 not only computationally lightweight in terms of parameters but also highly efficient after quantization, especially with Float16, which delivers the best speedup among all models.

EfficientNetB0, being the most complex of the three models, also benefits considerably from quantization in terms of inference time. Its dynamic range inference time is 35.45 ms, which decreases to 23.68 ms with Float16 and 23.6 ms with INT8 quantization. These reductions are substantial, bringing the model’s latency closer to that of the smaller MobileNet models while maintaining high accuracy, particularly in the Float16 format. Notably, the inference time for INT8 and Float16 quantization is nearly identical in EfficientNetB0, suggesting that hardware acceleration or model structure may influence the runtime advantages.

From the above results, quantization significantly reduced inference time across all models, with Float16 consistently offering the best or near-best performance gains. These improvements make quantized models highly attractive for real-time applications on resource-constrained devices. MobileNetV2 and EfficientNetB0 particularly stand out due to their excellent balance of speed and accuracy after quantization

|  |
| --- |
|  |
| **Figure 7: Quantization Inference Measures** |
|  |

|  |  |  |  |
| --- | --- | --- | --- |
|  | Dynamic Inference Time (ms) | Float16 Inference Time (ms) | INT8 Inference Time (ms) |
| MobileNet | 25.36 | 19.13 | 21.27 |
| MobileNetV2 | 20.2 | 11.46 | 15.82 |
| EfficientNetB0 | 35.45 | 23.68 | 23.6 |
| Table 2: Quantization Inference Measures | | | |

The quantization strategies demonstrated notable reductions in model size. For MobileNet, the original model size of approximately 18.95 MB was substantially reduced through quantization. Dynamic range quantization shrank the model to just 3.77 MB, representing an approximate 80% reduction, which is highly beneficial for deployment on memory-constrained edge devices. Float16 quantization increased the model size to 7.19 MB, which, while larger than the dynamic version, still offers significant compression compared to the original. The INT8 quantized model came in at 3.9 MB, offering similar savings as dynamic quantization but with lower accuracy, as previously discussed.

MobileNetV2 follows a similar pattern. The dynamic range quantized model is reduced to 3.06 MB, and the INT8 quantized version is slightly larger at 3.26 MB, both yielding compact model sizes well-suited for edge deployment. Float16 quantization increased the size to 5.57 MB, which again is larger than the integer-based versions but still more efficient than the original 16.87 MB. The smaller base parameter count of MobileNetV2 likely contributes to its tighter model sizes after quantization.

EfficientNetB0, while being the most accurate model, shows a different trend due to its more complex architecture. Its original model size of 23.81 MB is reduced to 5 MB through dynamic range quantization and 5.36 MB using INT8 quantization, both resulting in excellent compression. However, the Float16 quantized model is significantly larger at 9.02 MB, which may limit its usability on ultra-low-resource devices despite its outstanding accuracy retention.

, dynamic range and INT8 quantization consistently produce the smallest models, making them suitable for deployment on highly resource-constrained environments. Float16, while not always resulting in the smallest size, balances moderate compression with high accuracy, making it ideal when slightly more memory is available. EfficientNetB0, despite its superior accuracy, trades off model size and suffers heavily under INT8 quantization, which may limit its deployment without precision tuning.

|  |
| --- |
|  |
| **Figure 8: Quantization File Size Measures** |
|  |

|  |  |  |  |
| --- | --- | --- | --- |
|  | Dynamic Model Size (MB) | Float16 Model Size (MB) | INT8 Model Size (MB) |
| MobileNet | 3.77 | 7.19 | 3.9 |
| MobileNetV2 | 3.06 | 5.57 | 3.26 |
| EfficientNetB0 | 5 | 9.02 | 5.36 |
| Table 3: Quantization File Size Measures | | | |

6 DISCUSSION

**6.1 Resource Constraints**

A significant challenge encountered was the limitation in computational and memory resources during model development. I experimented with various models and quantization techniques, including Dynamic Range, Float16, and Full Integer quantization. However, each training and quantization cycle took over 40 minutes to complete, making it impractical to process multiple models concurrently. This limitation necessitated a highly manual workflow, where each model had to be trained, quantized, evaluated, and documented sequentially. As a result, the process of collecting results and generating analytical graphs for performance comparison required considerable time and effort, slowing down overall experimentation and analysis.

**6.2 XNNPack Compatibility Error**

While attempting to run a quantized version of MobileNetV3-Small and MobileNetV3-Large using TFLite with Full Integer quantization, an incompatibility error with the XNNPack delegate was encountered. The error indicated that both the MobileNetV3 versions are not fully supported by XNNPack when deployed with Full-Int quantization. A snapshot of this error is shown in Figure 9. XNNPack is a highly optimized delegate used to accelerate inference on CPUs, but its limited support for certain layers or quantization modes in specific models results in failure or fallback to slower interpreters. This necessitated switching to either Dynamic range or F-16 quantization strategy or relying on a CPU-only fallback, which reduced the benefits of model optimization. The lack of detailed documentation on delegate compatibility further complicated troubleshooting efforts.

|  |
| --- |
|  |
| **Figure 9: Snapshot of XNNPack Error** |

**6.3 Invalid Input Shape Due to Normalization**

A major challenge arose from incorrect preprocessing of input images during model training. Initially, normalization was performed by scaling pixel values between 0 and 1 using division by 255.0. However, this led to unexpected input shape and value mismatches, as many Keras models, such as MobileNetV2, expect input values to be standardized within the range of -1 to 1. Overlooking this detail in the early stages caused significant delays in the project timeline. For instance, MobileNetV2 initially achieved only 44% classification accuracy, far below expected benchmarks outlined in the TensorFlow documentation. The root cause was that each model required a different preprocessing function tailored to its architecture, a detail not clearly specified in the official documentation. This issue was resolved after a thorough review of Keras documentation, where appropriate preprocessing functions (e.g. “keras.applications.mobilenet\_v2.preprocess\_input”) were applied individually for each model, ultimately restoring model performance and ensuring consistency.

7 GITHUB REPOSITORY

You can find the training process and results in the Jupyter Notebook available in the provided GitHub repository.

8 CONCLUSIONS

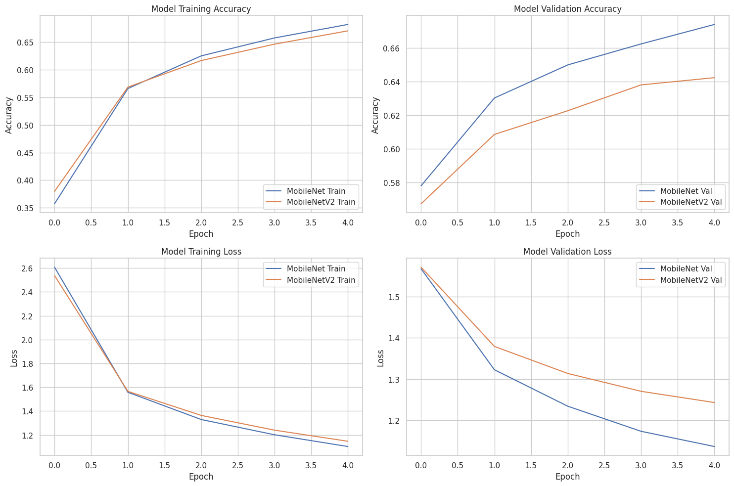
In this project, three lightweight convolutional neural network models MobileNet, MobileNetV2, and EfficientNetB0 were evaluated and compared on CIFAR-100 dataset to assess their performance, efficiency, and suitability for edge-device deployment. After training the models, three post-training quantization techniques—Dynamic Range, Float16, and Full Integer (INT8) were applied to reduce model size and improve inference speed. The results showed that EfficientNetB0 achieved the highest original accuracy (72.15%) and demonstrated strong resilience to Float16 quantization, while MobileNetV2 emerged as the most efficient in terms of both parameter count and inference time after quantization. Float16 quantization consistently preserved accuracy while improving inference speed, making it the most balanced choice for real-time applications. In contrast, INT8 quantization offered the highest compression but significantly reduced model accuracy, especially in more complex architectures.

For future work, there are still challenges to address. Firstly, due to time and resource constraints, models were only trained for five epochs; running more epochs with early stopping could enhance model generalization and accuracy. Secondly, fine-tuning the quantized models could potentially recover performance lost during aggressive quantization. Lastly, XNNPack compatibility issues prevented the successful quantization of MobileNetV3 models using INT8. Resolving this issue and extending the study to include MobileNetV3-Small and MobileNetV3-Large would further strengthen the findings and broaden the range of deployable models.

REFERENCES

1. Shuvo, M.M.H., Islam, S.K., Cheng, J. and Morshed, B.I., 2022. Efficient acceleration of deep learning inference on resource-constrained edge devices: A review. *Proceedings of the IEEE*, *111*(1), pp.42-91.
2. Careem, R., Johar, G. and Khatibi, A., 2024. Deep neural networks optimization for resource-constrained environments: techniques and models. *Indonesian Journal of Electrical Engineering and Computer Science*, *33*(3), pp.1843-1854.
3. Lee, H., Lee, N. and Lee, S., 2022. A method of deep learning model optimization for image classification on edge device. *Sensors*, *22*(19), p.7344.
4. Yuan, H., Cheng, J., Wu, Y. and Zeng, Z., 2022. Low-res MobileNet: An efficient lightweight network for low-resolution image classification in resource-constrained scenarios. *Multimedia Tools and Applications*, *81*(27), pp.38513-38530.
5. Debasish Kalita, 2025 Basics of CNN in Deep Learning. Analytics Vidhya.
6. codebasics (2021). *Quantization in deep learning | Deep Learning Tutorial 49 (Tensorflow, Keras & Python)*. [online] YouTube. Available at: https://www.youtube.com/watch?v=v1oHf1KV6kM [Accessed 14 Apr. 2025].
7. Team, K. (n.d.). *Keras documentation: Keras Applications*. [online] keras.io. Available at: https://keras.io/api/applications/.
8. ‌TensorFlow. (n.d.). *Transfer learning with a pretrained ConvNet | TensorFlow Core*. [online] Available at: https://www.tensorflow.org/tutorials/images/transfer\_learning.
9. ‌Kulkarni, U., Meena, S.M., Gurlahosur, S.V. and Bhogar, G., 2021. Quantization friendly mobilenet (qf-mobilenet) architecture for vision based applications on embedded platforms. *Neural Networks*, *136*, pp.28-39.
10. Chung, C.C., Chen, W.T. and Chang, Y.C., 2020, February. Using quantization-aware training technique with post-training fine-tuning quantization to implement a MOBILENET hardware accelerator. In *2020 Indo–Taiwan 2nd International Conference on Computing, Analytics and Networks (Indo-Taiwan ICAN)* (pp. 28-32). IEEE.
11. Kluska, P. and Zięba, M., 2020. Post-training quantization methods for deep learning models. In *Intelligent Information and Database Systems: 12th Asian Conference, ACIIDS 2020, Phuket, Thailand, March 23–26, 2020, Proceedings, Part I 12* (pp. 467-479). Springer International Publishing.

Appendix



|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Model | Test Accuracy | Test Loss | Parameters  (M) | Training Time (s) | Inference Time (s/sample) | Model Size (MB) |
| MobileNet | 0.66 | 1.13 | 3.80 | 372.53 | 4.27 | 18.95 |
| MobileNetV2 | 0.63 | 1.24 | 2.94 | 334.69 | 7.6 | 16.87 |
| EfficientNetB0 | 0.72 | 0.94 | 4.7 | 423.58 | 0.2052 | 23.81 |

