POLITECNICO DI TORINO

Master degree course in Electronic Engineering

Master Degree Thesis

Efficient Mixed-Precision Quantization of Deep Neural Networks for Edge Applications

Advisors
Prof. Mario Roberto Casu
M.Sc. Edward Manca
Dr. Luca Urbinati

Candidato
Yuliang CHEN

Academic Year 2023-2024

Summary

Over the past decade, deep learning has significantly advanced artificial intelligence (AI), revolutionizing domains such as speech recognition and image classification. However, the increased performance of deep neural networks (DNNs) comes with heightened computational complexity and energy demands, making deployment on resource-constrained devices, like embedded systems, challenging. TinyML addresses these limitations by optimizing models to balance accuracy and computational efficiency, especially within the Internet of Things (IoT) ecosystem. By enabling training and inference on edge devices, TinyML enhances data privacy, reduces energy consumption, and lowers latency. To achieve these goals, it employs techniques like precision reduction through quantization and designs more compact architectures to alleviate computational strain. This study explores QKeras, an open-source quantization library, using its "po2" mode to quantize six different neural network architectures, including two with mixed-precision quantization. The "po2" mode replaces traditional multiplication and division with bitwise shifts, reducing computational time and resource usage while maintaining a minimal accuracy drop of about 1% compared to the standard "auto" mode. Despite these benefits, some networks faced challenges during training, including increased sparsity, complexities that will be examined in detail.

Acknowledgements

First of all, I am deeply grateful to Professor Mario Casu for accepting me to work on this project. I would also like to extend my heartfelt appreciation and gratitude to Luca and Edward for their patient and meticulous guidance. Every piece of advice from them has greatly benefited me. Without their help, I would not have been able to complete this task. I am truly thankful for all of this.

Contents

Chapter 1

Fundamentals of Neural Networks

1.1 Overview of Artificial Neural Networks

1.1.1 General Intuition

Artificial Neural Networks (ANNs) are computational models, their design inspired by the structure and function of biological neural networks.[1] These models excel at recognizing images and patterns and relationships in data, particularly in tasks as classification, regression, and pattern recognition. ANNs consist of interconnected layers of units, called neurons, which process information in a manner similar to how human neurons respond to stimuli. The primary goal of an ANN is to learn from data by adjusting its internal parameters, allowing it to make accurate predictions or decisions. This learning process is iterative, with the network progressively refining its representations and parameters to reduce errors and improve accuracy. ANNs are distinguished by their ability to model non-linear,complicated relationships, which sets them apart from traditional linear models and makes them a powerful tool in various research and application domains.

1.1.2 Neurons (Artificial Neurons)

The artificial neuron is the fundamental unit of an artificial neural network, modeled after the biological neurons found in the human brain. These artificial neurons.,Each neuron receives one or more inputs, processes them, and produces an output[2].to enable the network to process and interpret data, allowing it to evolve and make informed predictions. Through these neurons, the network learns by adjusting weights and biases, gradually enhancing its performance and accuracy. In a neural network, inputs are represented as a set of features, each associated with a weight that indicates its relative significance. The neuron functions as a processor, multiplying each feature by its corresponding weight and summing the results. Once the sum is computed, the neuron applies an activation function, introducing non-linearity to the network. This critical step allows the network to model complex and intricate relationships within the data, which would be unattainable with linear models alone. The introduction of non-linearity enhances the network's ability to capture more sophisticated patterns and dependencies in the data.

1.1.3 Activation Functions

Activation functions play a crucial, yet often understated role in neural networks, determining whether a neuron should activate and contribute to the network's output or remain inactive. By applying a mathematical transformation to the weighted sum of inputs, these functions introduce the essential non-linearity required for the network to capture complex relationships in the data. This non-linearity enables neural networks to move beyond the limitations of linear models, allowing them to detect and model intricate patterns. Without activation functions, even the most complex neural networks would be reduced to simple linear operations, limited in their ability to represent complex data. It is through activation functions that neural networks achieve the flexibility and depth necessary to function as powerful learning systems
sigmoid function
The sigmoid function is a commonly utilized activation function in NNs, known for its ability to map any input, regardless of magnitude, into a value between 0 and 1. This characteristic makes it particularly effective for tasks that involve probability-based outcomes, such as binary classification. By smoothly transforming inputs into a bounded range, the sigmoid function provides a clear interpretation of outputs as probabilities. However, one of its limitations is its tendency to produce outputs that are close to either 0 or 1 for large or small inputs, which can lead to the vanishing gradient problem. This issue reduces the effectiveness of weight updates during training, thereby slowing the learning process and impeding the network's ability to converge efficiently.

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

The sigmoid function is smooth and differentiable, which makes it suitable for gradient-based optimization techniques like backpropagation.[3]However, the sigmoid function is not without its limitations. It is particularly susceptible to the vanishing gradient problem, a common issue in deep networks. When input values deviate significantly from zero, the gradients become exceedingly small, diminishing almost to the point of ineffectiveness. This drastically reduces the speed at which the network learns, as the gradients fail to provide sufficient updates to the model's weights. Consequently, the learning process slows down considerably, making it challenging for the model to converge efficiently and update its parameters in a meaningful way. This issue can significantly hinder the overall performance of the network, especially in deeper architectures.

ReLU (Rectified Linear Unit)

ReLU is a piecewise linear function that outputs the input directly if it is positive; otherwise, it outputs zero.[4] The Rectified Linear Unit (ReLU) has become the predominant activation function in deep learning due to its simplicity and efficiency of computations. Its primary advantage lies in its ability to avoid the vanishing gradient problem, as it allows positive gradients to pass through unaffected, thereby accelerating the training process. This characteristic makes ReLU highly effective in deep networks, as it enables faster convergence compared to other activation functions. However, ReLU is not without its drawbacks. One notable issue is the "dead neuron" problem, where certain neurons become inactive and consistently output zero for any input. This occurs when neurons are driven into a state of permanent dormancy, often due to poor weight initialization or an overly aggressive learning rate. Once neurons enter this inactive state, they no longer contribute to the learning process, which can negatively impact the network's performance. The Rectified Linear Unit (ReLU) has become the predominant activation function in deep learning due to its simplicity and computational efficiency. Its primary advantage lies in its ability to avoid the vanishing gradient problem, as it allows positive gradients to pass through unaffected, thereby accelerating the training process. This characteristic makes ReLU highly effective in deep networks, as it enables faster convergence compared to other activation functions. However, ReLU is not without its drawbacks. One notable issue is the "dead neuron" problem, where certain neurons become inactive and consistently output zero for any input. This occurs when neurons are driven into a state of permanent dormancy, often due to poor weight initialization or an overly aggressive learning rate. Once neurons enter this inactive state, they no longer contribute to the learning process, which can negatively impact the network's performance.

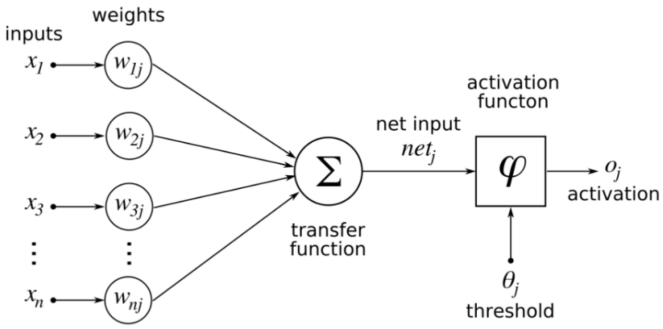


Figure 1.1: the structure of an artificial neuron.[5]

1.1.4 Structure of a Neural Network

A neural network is typically composed of three types of layers

- **Input Layer:**

Receives the input features. Each neuron in this layer represents one feature of the input data.[6]

- **Hidden Layer(s):**

These are the tireless workers of the network, nestled between the input and output layers. They carry out the bulk of the computational effort, steadily guiding the network as it learns to recognize patterns and uncover features hidden within the data.

- **Output Layer:**

The grand finale, where the network's efforts culminate. This layer delivers the ultimate result, be it a class label in classification tasks or a predicted value for regression. All the learning converges here, transforming computation into meaningful output.

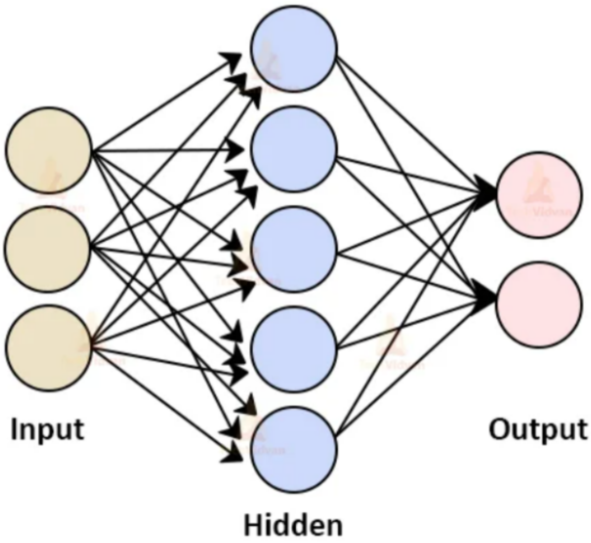


Figure 1.2: the structure of a Neural Network.[7]

Consider a simple network with three input neurons, a hidden layer comes with five neurons, and two output neurons. Each connection between these neurons carries a weight, which is fine-tuned during training section,in order to improve the network's performance.

1.1.5 Loss Function

The loss function acts as the model's truth-teller, revealing just how far the predicted outputs veer from the true target values (the ground truth). In essence, it quantifies the error between the model's predictions and reality. Its mission is clear: to guide the learning process by signaling how to adjust the model's parameters—weights and biases—so that this error gradually decreases. A lower loss score is the model's way of saying it's on the right track, inching ever closer to making accurate predictions.

- **Mean Squared Error (MSE)**

MSE is commonly employed in regression tasks, where the aim is to predict continuous values.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Mean Squared Error (MSE) computes the average of the squared differences between the predicted values $\hat{y}_i$ and the actual values y_i. By squaring the differences, MSE gives greater emphasis to larger errors, meaning that predictions far off the mark (outliers) carry a heavier weight in the overall loss. This sensitivity can be both a blessing and a curse—while it helps in identifying major errors, it can also overreact to outliers, which may either improve or skew the model's learning, depending on the nature of your data.
MSE provides a smooth gradient, making it a popular choice for gradient-based optimization algorithms like gradient descent. However, its sensitivity to outliers can become a double-edged sword. A few extreme data points may disproportionately influence the model, potentially leading to skewed results.

- **Cross-Entropy Loss**

Cross-Entropy Loss, or log loss, is the go-to loss function for classification tasks, especially when the model outputs probabilities, such as in softmax-based multiclass classification. This loss function is well-suited for predicting class membership and also takes into account the model's confidence in those predictions, making it invaluable in tasks where probability plays a crucial role.

$$\text{Cross-Entropy Cost} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{c=1}^{C} y_{i,c}\log(\hat{y}_{i,c})$$

This formula is used to calculate the average loss of the model in multi-class tasks. By weighting the predictions for each sample across all categories (with the actual category weighted as 1 and other categories weighted as 0), it effectively evaluates the accuracy of the model's predictions. The purpose of the cross-entropy loss function is to encourage the model to increase the prediction probability for the correct category while reducing the prediction probability for incorrect categories. Cross-Entropy Loss measures the divergence between two distributions—the true distribution (the actual labels) and the predicted one (the model's outputs). In the world of binary classification, this loss is calculated for each class separately; in multi-class classification, it spans across all classes. When the model, in its folly, places high confidence in an incorrect class, the loss function exacts a harsh penalty, reminding the model of its misjudgment. Cross-Entropy Loss is particularly well-suited for classification tasks, as it ensures that the predicted probabilities align closely with the true labels. This loss function encourages the targeted model to make sharper and more confident predictions, which is especially important when the accuracy of probabilities is critical. One of its key advantages is that it provides a strong gradient, even in cases of small errors, which accelerates the learning process by allowing the targeted model to adjust more effectively during training.

1.1.6 Cost Function

While the loss function measures the model's performance on a single data point, the cost function extends this assessment across the entire dataset. By summarizing the model's overall performance, the cost function serves as the objective to be minimized during training. The goal of reducing the cost function is to systematically decrease prediction errors, thereby refining the model's generalization ability and improve accuracy.

1.1.7 Minimizing Cost with Gradient Descent

Gradient Descent is a widely used optimization technique aimed at minimizing the Cost function. This method iteratively adjusts the network's weights by calculating the gradient of the loss function for each weight. By making incremental updates in the direction which reduces the loss, Gradient Descent gradually guides the model toward optimal weights, ultimately improving the accuracy of its predictions over time.
the steps are below

- **Calculate the Gradient**

The gradient, or the rate of change of the loss function with respect to each weight parameter, is calculated. This shows how the loss would change if we adjusted each weight slightly, guiding the network on how to update its parameters to reduce the overall error.Update Weights: The weights are adjusted in the opposite direction of the gradient.[8] This adjustment is governed by the learning rate, a vital force that determines how grand or slight the weight updates shall be. A well-chosen learning rate strikes the perfect balance—neither rushing ahead with reckless abandon nor creeping along too cautiously.

- **Repeat**

This cycle of optimization continues, with the process repeating iteratively until the loss function descends to its minimum, or until additional efforts result in diminishing returns. The model continuously adjusts and refines itself, learning with each iteration, until no further significant improvements can be made—marking the completion of its journey toward precision and optimal performance.

- **Final Goal**

The ultimate aim in training a neural network is to uncover the optimal set of weights that minimizes the loss function, thus enabling the model to make highly accurate predictions on previously unseen data. This goal forms the cornerstone of how neural networks learn and adapt, guiding the model toward improved performance with every iteration. The process is one of continual refinement, where the network's accuracy steadily increases until it reaches its most effective form.

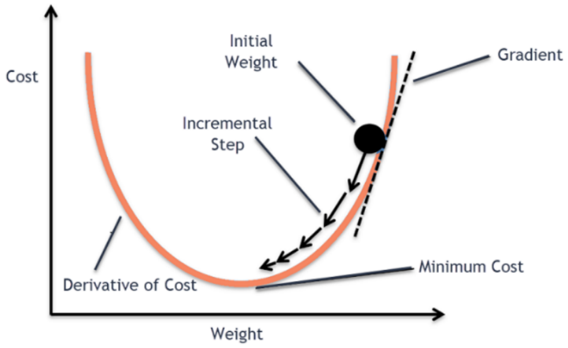


Figure 1.3: gradient descent algorithm.[9]

1.2 Deep Neural Networks (DNNs):

This is a sample for a bullet list.

1.2.1 Introduction to DNNs

Deep Neural Networks (DNNs) represent the advanced evolution of Artificial Neural Networks (ANNs), characterized by their significantly deeper and more intricate architectures. While ANNs may operate with a limited number of layers, DNNs go further, employing multiple hidden layers to unlock the latent complexity within data. This increased depth grants DNNs the ability to tackle large, complex datasets, capturing subtle patterns and nuances that simpler models might overlook. As a result, DNNs are highly effective in a wide range of sophisticated applications, including image recognition, natural language processing, and predictive analytics. Their depth and complexity make them invaluable tools in modern machine learning, allowing them to deliver precise and nuanced insights into the most intricate of data structures. At their core, DNNs follow the same principles as ANNs, consisting of layers of interconnected neurons. Each neuron processes inputs, applies an activation function, and passes its output to the next [10] The difference which matters most, between Deep Neural Networks (DNNs) and simpler networks lies in their depth. Each hidden layer in a DNN goes deeper into the data, finding more abstract and complex features. This layered structure allows DNNs to create detailed and rich representations, giving them the ability to discover complex, non-linear relationships that simpler networks cannot handle. Training a DNN is similar to training an Artificial Neural Network (ANN), where weights are adjusted using techniques like gradient descent. However, as DNNs get deeper, they become more complex. They face challenges like vanishing and exploding gradients, which can slow down or stop learning. To overcome these issues, more advanced optimization methods and regularization techniques are used to help the network not only reduce errors but also improve its performance on new, unseen data.

1.2.2 Convolutional Neural Network

Introduction

Convolutional Neural Networks (CNNs) are powerful deep learning models, particularly designed to excel in analyzing visual data. For example, images and videos. Their primary goal is to enable machines to recognize and interpret patterns within this data, making them essential for tasks like image classification, facial recognition, and object detection etc.
What sets CNNs apart is their capacity to autonomously learn and extract relevant features from raw input, removing the necessity for manual feature selection that is often required by conventional models. In the initial layers, CNNs identify basic features such as edges, corners, and textures, which serve as the foundation for more complex structures.
As data progresses through the network, CNNs refine their feature extraction, detecting increasingly sophisticated patterns. These deeper layers integrate simple features to recognize more complex elements, such as shapes, objects, and even specific components within those objects. By the time data reaches the final layers, the CNN has developed a comprehensive understanding of its input, enabling it to make accurate classifications or predictions based on the processed images.
This hierarchical approach to feature extraction allows CNNs to efficiently grasp spatial relationships in visual data. Their ability to learn both fundamental and advanced features across multiple layers of abstraction has positioned CNNs as a dominant force in computer vision, driving advancements in areas like image recognition, medical imaging, and autonomous systems.

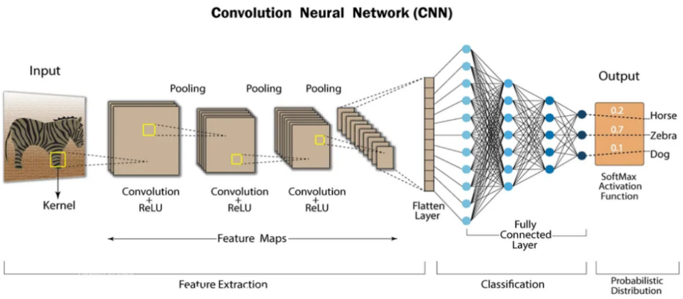


Figure 1.4: the structure of CNNs Network.[11]

it is the typical architecture of a CNN model, composed of three main types of layers:

- Convolutional layers
- Pooling layers

- Fully-connected layers

Convolutional Layer

The convolutional layer is the essential component of the whole Convolutional Neural Networks (CNNs), enabling them to process visual data with remarkable precision and efficiency. Unlike fully connected layers, where each unit is connected to every other, the convolutional layer focuses on specific, localized fields of the input. It applies a set of learnable filters to these regions, identifying patterns within the data—such as those found in images—through a process called convolution. In this layer, small filters or kernels, typically 3x3 or 5x5 in size, move across the input data, performing convolution operations. At each position, the filter interacts with a small portion of the input, producing a feature map that highlights key patterns, such as edges, corners, and textures. These feature maps are crucial, as they represent the essential details the CNN uses to build its understanding of the data. Through this process, the convolutional layer begins to unravel the complexities within the input, laying the groundwork for further layers to build a more comprehensive interpretation of the visual information.
Several key parameters define the behavior of a convolutional layer:

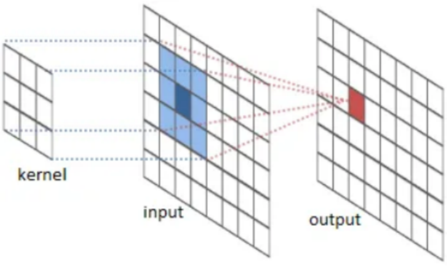


Figure 1.5: Image convolution.[12]

- **Filters (Kernels)**

Filters, also known as kernels, serve as the essential tools in a convolutional layer, moving across the input data to uncover specific features, much like explorers seeking valuable insights within complex patterns. Each filter is created to detect particular elements, such as sharp edges or detailed textures, by analyzing small sections of the input at a time. The values within these filters are learned during training, gradually refined to better identify the key characteristics of the data. As more filters are applied within a layer, the network's ability to detect a wide range of patterns increases, enriching its understanding and representation of the input. Each filter contributes to the network's knowledge, enhancing its ability to recognize the subtle features that drive accurate predictions and classifications.

- **Stride**

The stride determines the movement of the filter across the input data, dictating how far the filter shifts with each step. A stride of 1 moves the filter one pixel at a time, capturing fine details and ensuring a thorough examination of the data. Larger strides, such as 2 or 3, cover more ground with each step, reducing the spatial dimensions of the output while focusing on broader patterns. This balance between precision and efficiency allows the network to capture either minute details or more general features, depending on the stride chosen.

- **Padding**

Padding involves adding extra pixels around the borders of the input data to ensure that no information is lost at the edges. In "same" padding, pixels are added to maintain the input and output dimensions, allowing the filter to process edge data without shrinking the output. In contrast, "valid" padding adds no extra pixels, leading to a smaller output as the filter moves within the existing boundaries of the input. Both forms of padding play a critical role in preserving important information and ensuring that all areas of the input are processed.
Through the combined use of shared filters and these techniques, the convolutional layer achieves a significant reduction of the number of parameters, in comparison to fully connected layers. This efficiency helps keep the model from overfitting and enables the model to be both lightweight and powerful, striking a balance between speed and accuracy. the output size of a feature map is converted by the parameters mentioned as:

$$\text{Output Size} = \frac{\text{Input Size} - \text{Filter Size} + 2 \times \text{Padding}}{\text{Stride}} + 1$$

Pooling Layers

Pooling layers play a crucial role in Convolutional Neural Networks (CNNs), tasked with reducing the spatial dimensions of the feature maps generated by previous layers. This reduction helps to lower the computational demands of the network while enhancing its robustness to slight variations or distortions in the input data. Pooling operates by dividing the feature map into smaller, non-overlapping regions and summarizing each with a single value. Max pooling, the most widely used method, selects the highest value from each region, ensuring that the most prominent features are preserved while discarding less significant details. In contrast, average pooling calculates the average value of each region, creating a more smoothed representation by retaining generalized information.
Max pooling, however, is favored in tasks such as object recognition, as it retains the most important features critical for classification
Key parameters governing the pooling process include:

- Pool Size: The dimensions of the region being pooled.
- Stride: The step size or distance by which the pooling window moves across the feature map.
- Padding: Seldom used in pooling, as the goal is typically to reduce spatial dimensions rather than preserve the original size.

Fully Connected Layers

Fully connected layers, so called as dense layers, mark the final stage in the journey of a Convolutional Neural Network. After convolutional and pooling layers have extracted and condensed meaningful patterns from the input data, fully connected layers interpret these features and make the network's final predictions.

In a dense layer, every neuron is linked to all the neurons in the layer before it, ensuring that all the extracted information is considered. These layers take the high-level features identified by earlier layers and translate them into the final output, whether that be classification labels in a classification task or predicted values in regression. Thus, dense layers serve as the decisive interpreters of the network's learning, transforming it into actionable predictions or classifications.

Depthwise Convolutional Layers

In traditional convolutional layers, a single filter operates across all input channels simultaneously, generating a single output feature map per filter. Depthwise convolutional layers, however, take a more specialized approach. Each input channel is paired with its own dedicated filter, a process referred to as depthwise convolution.

In a depthwise convolutional layer, each input channel is assigned an individual filter that works independently over the spatial dimensions of the input. For an input with CCC channels, there are CCC filters, each focused on its respective channel. This results in CCC feature maps, one for each input channel, preserving the unique characteristics of each.

Following this, the pointwise convolutional layer comes into play. This is a 1x1 convolution that operates across all the channels, merging the features extracted by the depthwise convolution. Though the filter size is small, its role is significant, as it combines the depthwise outputs into a final set of feature maps with the desired number of channels. Together, these layers—depthwise and pointwise—compose the depthwise separable convolution, an efficient variation of standard convolutional layers. This method reduces computational complexity while maintaining network performance, offering a refined balance of efficiency and effectiveness in modern neural networks

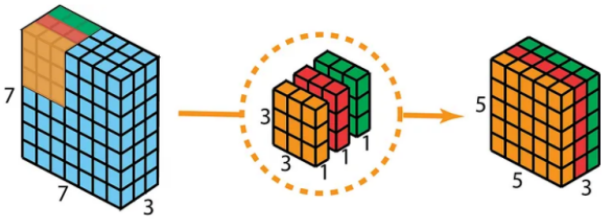


Figure 1.6: depthwise convolutional layer.[13]

the raito of Con2D layers and depthwise convolutional layers in transforming the input layer (H x W x D) into the output layer (H-h+1 x W-h+1 x Nc)[13], with Nc kernels of size h x h x D The computational cost of traditional convolution can be expressed as:[13]

$$\text{Computational Cost}_{\text{Conv2D Convolution}} = N_c \times h \times h \times D \times (H - h + 1) \times (W - h + 1)$$

where N_c is the number of convolution filters, $\mathbf{h \times h}$ is the size of the convolution kernel, **D** is the number of input channels, **H** and **W** are the height and width of the input image, respectively. This formula describes the total computational cost when applying N_c filters to each input channel.[13] In contrast, depthwise separable convolution splits the convolution operation into two steps: depthwise convolution and pointwise convolution.[14] The computational cost of depthwise convolution depends only on the number of input channels and the kernel size[13]

$$\text{Computational Cost}_{\text{Depthwise}} = D \times h \times h \times 1 \times (H - h + 1) \times (W - h + 1)$$

For pointwise convolution (also known as 1×1 convolution), the computational cost depends mainly on the number of filters and the number of channels:[13]

$$\text{Computational Cost}_{\text{Pointwise}} = N_C \times 1 \times 1 \times D \times (H - h + 1) \times (W - h + 1)$$

Thus, the total computational cost of depthwise separable convolution is the sum of the two:

$$\text{Total Computational Cost} = (h \times h + N_c) \times D \times (H - h + 1) \times (W - h + 1)$$

By comparing the computational costs of traditional convolution and depthwise separable convolution, we can derive the following ratio formula:[13]

$$\frac{1}{N_c} + \frac{1}{h^2}$$

This formula demonstrates that depthwise separable convolution can substantially lower the computational cost compared to traditional convolution. When the number of filters is large, depthwise separable convolution reduces the workload by breaking the operation into two steps. This separation allows for a significant improvement in computational efficiency while still preserving the model's accuracy. It's a smart balance between speed and precision.

1.3 Advanced Architectures

1.3.1 Rsidual Neural Networks (ResNets)

Residual Neural Networks, commonly known as ResNets, represent a significant breakthrough in the development of deep neural network architectures. Introduced by Kaiming He and his team in 2015, ResNets have become a pivotal structure in the research of deep learning, renowned for their ability to train extremely deep networks while improving both accuracy and stability.

A key challenge in training deep neural networks is the issue of vanishing or exploding gradients. This phenomenon occurs when gradients, essential for updating the network's weights during backpropagation, either diminish excessively or grow uncontrollably. As a result, the network may face difficulties in convergence, impeding effective training. In fact, merely increasing the number of layers often exacerbates the problem, leading to performance degradation, a problem known as the degradation issue.

ResNets address these challenges through the concept of residual learning. Rather than requiring each layer to learn a complete transformation from input to output, ResNets allow the network to focus on learning the residual function, which represents the difference between the input and the output. This is achieved using shortcut or skip connections that bypass one or more layers, directly adding the input to the output. By doing so, the network learns to refine or adjust the input, focusing on the residual elements, which significantly enhances both training efficiency and performance.
Mathematically, this is expressed as:

$$\text{Output} = F(x) + x$$

F(x) represents the residual function and x is the input passed through the shortcut A typical residual block consists of two or more convolutional layers, Each is followed by batch normalization and a ReLU activation function. The shortcut connection skips over these layers, allowing the network to learn the residual rather than the complete transformation. This architecture stabilizes the training process and improves performance as more layers are added.

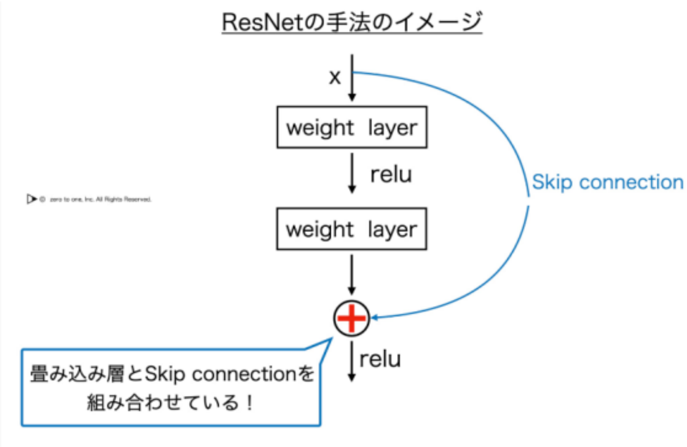

ResNetの手法のイメージ
x
weight layer
relu
weight layer
Skip connection
relu
畳み込み層とSkip connectionを組み合わせている！
© zero to one, Inc. All Rights Reserved.

Figure 1.7: the structure of ResNet.[15]

Residual Networks (ResNets) have become indispensable in the field of deep learning, especially in computer vision tasks, where they serve as the foundational architecture for numerous state-of-the-art models. Their successful implementation has also inspired further advancements, such as DenseNets and Highway Networks, which similarly leverage shortcut connections to facilitate learning in deep architectures. These innovations build upon the core principles of residual learning, enhancing the training efficiency and performance of increasingly complex neural networks.

1.3.2 MobileNetV2

MobileNetV2 is a pioneering deep learning architecture tailored for mobile and embedded devices, where computational resources and power efficiency are critical constraints. Building upon the foundation laid by MobileNetV1, MobileNetV2 introduces key advancements aimed at achieving an optimal balance between performance and computational efficiency, all while preserving high levels of accuracy. The central innovation of this architecture is the inverted residual block, which integrates depthwise separable convolutions with residual connections. This strategic design enables MobileNetV2 to deliver strong performance while being highly suitable for low-power devices, making it a preferred choice in resource-constrained environments.

Bottleneck Residual Block

The bottleneck residual block is a defining feature of MobileNetV2, specifically engineered to maximize computational efficiency while maintaining the network's capacity to tackle complex tasks. This block utilizes a "bottleneck" structure, where the dimensionality of the input is reduced before undergoing further processing, and then restored afterward. This approach is instrumental in minimizing the number of parameters, thereby reducing computational costs, while still preserving the network's ability to capture and learn detailed and intricate features effectively. Structure of the Bottleneck Block:

- **First 1x1 Convolution (Expansion Layer):**

The bottleneck residual block initiates with a 1x1 convolutional layer, referred to as the expansion layer, which expands the input dimensionality by a factor, often sixfold. This expansion enables the network to capture a richer set of complex features while maintaining a relatively low computational cost, enhancing both efficiency and performance. The strategic use of 1x1 convolutions exemplifies how significant advancements in network capability can be achieved through small, targeted operations, highlighting the effectiveness of this design in maximizing feature extraction with minimal resource expenditure.

- **Depthwise Convolution:**

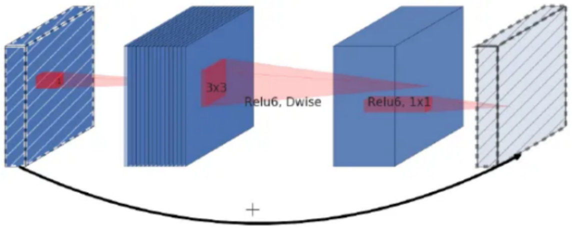
Once the input is expanded, it flows through a depthwise convolution layer, where each input channel is processed independently. A distinct filter is applied to each channel, allowing the extraction of spatial features while maintaining low computational complexity. This operation is central to MobileNetV2's efficiency, as it significantly reduces the number of operations required in comparison to standard convolutions, making it particularly suitable for resource-constrained environments without compromising on the extraction of spatial information.

- **Second 1x1 Convolution (Projection Layer):**

The final stage of the bottleneck block consists of a second 1x1 convolution, referred to as the projection layer, which compresses the expanded dimensionality back to its original size. This compression ensures that the critical learned information is retained in a more compact form, optimizing the data representation while reducing computational overhead. By restoring the input to its original dimensions, this step effectively prepares the data for the subsequent layers of the network, maintaining the balance between efficiency and feature learning.

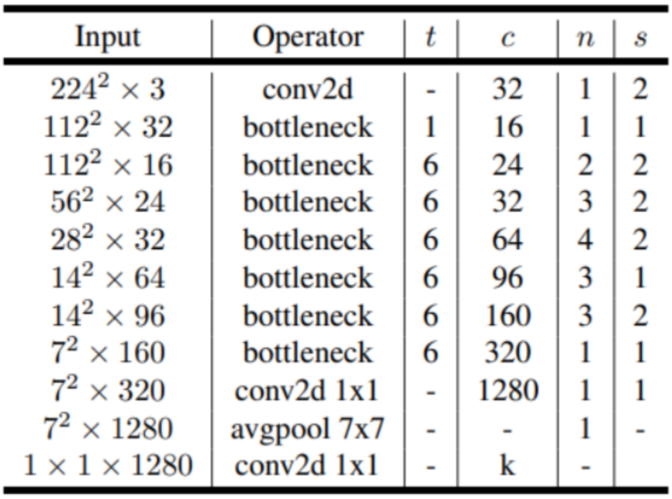
Residual Connection (Shortcut)

When the input and output dimensions are identical, a residual connection is introduced, similar to the structure utilized in ResNets. This shortcut allows the input to be directly added to the output, facilitating smoother gradient flow during training and improving the overall efficiency of the network. The inclusion of the residual connection ensures that the network preserves valuable information from earlier layers while simultaneously learning new features, enhancing both the stability and learning capacity of the model.



An inverted residual block connects narrow layers with a skip connection while layers in between are wide

Figure 1.8: An inverted residual block connects narrow layers with a skip connection, while the layers in between are wide [16]



Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-	

Figure 1.9: whole architecture of mobilenetV2 [17]

The specific architecture of MobileNetV2 is illustrated in the chart, distinguished by its innovative inverted residual blocks. These blocks, in conjunction with residual connections, allow the network to capture fine-grained and intricate features while optimizing computational efficiency. At the core of this design is the use of depthwise separable convolutions, which decompose traditional convolution operations into smaller, more efficient steps. This decomposition significantly reduces computational overhead, making MobileNetV2 far more resource-efficient compared to conventional models, all while maintaining the network's capacity to accurately interpret and process complex data.

1.3.3 EfficientNetB0

EfficientNet marks a substantial breakthrough in neural network architecture, achieving an ideal balance between high performance and computational efficiency. Building on the foundation of MobileNetV2, which emphasizes inverted residual blocks and depthwise separable convolutions, EfficientNet enhances this design with its innovative compound scaling method. This approach systematically scales the network across multiple dimensions—depth, width, and resolution—enabling more effective resource utilization while ensuring that the model grows in a balanced and optimized manner to improve accuracy without excessive computational costs.

Traditionally, neural networks have been scaled by independently increasing depth (adding more layers), width (increasing the number of neurons per layer), or resolution (processing higher-resolution input images). However, scaling along just one of these dimensions often results in models that are inefficient and disproportionately large, yielding only marginal gains in performance. EfficientNet addresses this limitation through compound scaling, harmoniously adjusting depth, width, and resolution in a unified approach. This balanced approach ensures that the model grows efficiently across all dimensions, leading to improved performance without unnecessary computational overhead.

EfficientNet builds upon the robust foundation laid by MobileNetV2, leveraging inverted residual blocks and depthwise convolutions. However, it introduces key enhancements, such as compound scaling and optimized architecture search, that further elevate both performance and efficiency. These refinements allow EfficientNet to maintain computational parsimony while achieving superior accuracy, making it particularly suited for resource-constrained environments without sacrificing model effectiveness. Here are some key features of EfficientNet:

- **Inverted Residual Blocks:**

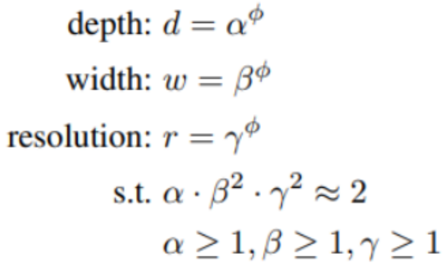
Similar to MobileNetV2, EfficientNet incorporates inverted residual blocks with linear bottlenecks to preserve the model's compactness. This architectural choice effectively minimizes the number of parameters and computational overhead, enabling the model to capture intricate patterns with high efficiency. By maintaining a streamlined structure, EfficientNet is able to model complex data representations without unnecessary increases in size or computational complexity, ensuring both scalability and performance.

- **Swish Activation Function:**

Instead of the traditional ReLU activation, EfficientNet utilizes the Swish activation function, which facilitates smoother gradient propagation, particularly in deeper layers of the network. This activation function enhances both accuracy and learning efficiency, allowing the model to achieve superior performance while maintaining stability throughout the training process.

- **Compound Scaling:**

The hallmark innovation of EfficientNet lies in its compound scaling technique, which simultaneously adjusts the network's depth, width, and resolution. This balanced scaling approach mitigates the inefficiencies associated with the independent scaling methods traditionally employed, ensuring a more cohesive and efficient model growth. Consequently, EfficientNet demonstrates outstanding performance across diverse tasks, such as image classification and object detection, while maintaining a high degree of computational efficiency.
the method of compound scaling set up some constrains about

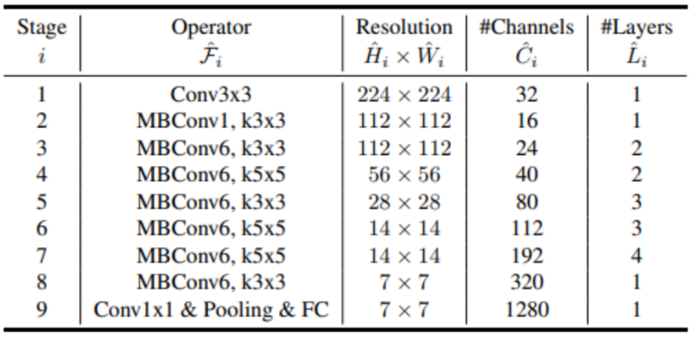


$$\begin{aligned} \textbf{depth: } & d = \alpha^{\phi} \\ \textbf{width: } & w = \beta^{\phi} \\ \textbf{resolution: } & r = \gamma^{\phi} \\ \textbf{s.t. } & \alpha \cdot \beta^2 \cdot \gamma^2 \approx 2 \\ & \alpha \geq 1, \beta \geq 1, \gamma \geq 1 \end{aligned}$$

Figure 1.10: Compound scaling equation.[18]

A grid search can be used to systematically explore different values of α, β, and γ. Intuitively, ϕ is a user-specified coefficient that controls how many more resources are available,for model scaling.[19] Following the grid search, the baseline model is meticulously constructed by combining manual design principles with advanced techniques like Neural Architecture Search (NAS). For each variant of EfficientNet—whether B1, B2, B3, or beyond—a comprehensive search process identifies the optimal scaling factors: α (depth), β, (width), and γ (resolution). These three factors dictate how the network grows across different dimensions as ϕ, the compound scaling coefficient, changes, ensuring that the model expands gracefully and efficiently.

EfficientNet doesn't just offer a single model but rather a family of models (B0, B1, B2, etc.), each precisely scaled to accommodate varying computational requirements and performance demands. This versatility allows practitioners to select the most suitable model based on specific resource constraints and application needs, making EfficientNet an adaptable and powerful architecture across a wide array of use cases.



Stage i	Operator $\hat{\mathcal{F}}_i$	Resolution $\hat{H}_i \times \hat{W}_i$	#Channels $\hat{C}_i$	#Layers $\hat{L}_i$
1	Conv3x3	224×224	32	1
2	MBConv1, k3x3	112×112	16	1
3	MBConv6, k3x3	112×112	24	2
4	MBConv6, k5x5	56×56	40	2
5	MBConv6, k3x3	28×28	80	3
6	MBConv6, k5x5	14×14	112	3
7	MBConv6, k5x5	14×14	192	4
8	MBConv6, k3x3	7×7	320	1
9	Conv1x1 & Pooling & FC	7×7	1280	1

Figure 1.11: EfficientNet-B0 baseline network[20]

In this context, EfficientNet B0 is chosen as the model of preference. It strikes an optimal balance between computational efficiency and performance, providing a high degree of precision and capability without the extensive resource requirements of its larger counterparts. EfficientNet B0 is ideally suited for a broad spectrum of tasks, delivering robust results while minimizing the strain on system resources. As a harmonious blend of power and subtlety, it excels in applications that require both high performance and efficient resource utilization, making it a versatile and reliable option for various computational challenges.

Chapter 2

Neural Network Quantization

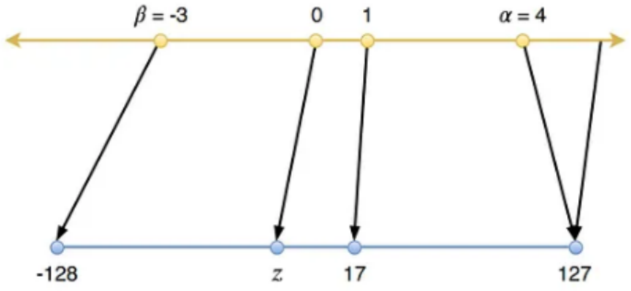
2.1 Basics of Quantization in Neural Networks

Quantization in neural networks serves the purpose of making models more efficient by reducing the precision of numerical values, such as weights and activations, from high-precision formats like 32-bit floating-point to lower precision formats, such as 8-bit integers. This reduction significantly lowers both the memory footprint and computational demands, which is especially important for deploying models on edge devices or in environments with limited resources, such as mobile phones and embedded systems. Despite the risk of slight accuracy loss due to reduced precision, techniques like Quantization-Aware Training (QAT) enable the model to learn and adapt during training, minimizing this degradation. The ultimate goal is to find a balance between performance and resource efficiency, allowing neural networks to be more widely deployed in real-world, resource-constrained applications.

2.2 Overview of Affine Quantization Schemes

Affine quantization schemes, akin to the methodical process of refinement, convert floating-point representations into a more efficient integer domain. This transformation significantly reduces computational load and memory usage, thereby enhancing the overall efficiency of deep learning models. By applying a linear transformation to the data through the use of a scaling factor and offset, high-precision floating-point values are mapped into lower-precision integer formats.

Crucially, this approach retains the linear relationships inherent in the original data, ensuring that the essential characteristics are preserved despite the reduction in precision. The primary objective of affine quantization is to minimize computational complexity while maintaining the integrity of the data. This enables neural networks to function effectively in low-resource environments, allowing models to remain lightweight and computationally efficient without sacrificing performance or compromising the fidelity of the information they encode.



(a) Affine quantization

Figure 2.1: affine quantization illustration.[21]

Let's break down the affine quantization formula step by step.

1. **Determine the Maximum and Minimum Values of the Range**

   The first step in affine quantization is to determine the range of the input data (floating-point values) and the maximum and minimum values of the quantized integer range.
   α The minimum value of the original data
   β The maximum value of the original data
   α_q The minimum value of the quantized data
   β_q The maximum value of the quantized data

2. **Calculate the Scaling Factor**

   The scaling factor s is used to map the floating-point data range to the quantized integer range. It is calculated as:

   $$s = \frac{\beta - \alpha}{\beta_q - \alpha_q}$$

3. **Calculate the Zero-Point** The zero-point z ensures that the zero in the original data is correctly mapped to the quantized integer space. The formula is:

   $$z = \text{round}\left(\frac{\beta_q \alpha - \alpha_q \beta}{\beta - \alpha}\right)$$

4. **Define Clip Function** The clip function is defined as:

$$\text{clip}(x,l,u)=\begin{cases} l & \text{if } x<l \\ x & \text{if } l\le x\le u \\ u & \text{if } x>u \end{cases}$$

The clip function ensures that the quantized result remains within the defined quantization range and does not exceed the given upper and lower bounds.

5. **Quantization Formula** Using the scaling factor and zero-point the input data and clip function, x is quantized into an integer value as follows:

$$f_q(x,s,z)=\text{clip}\left(\text{round}\left(\frac{1}{s}x+z\right),\alpha_q,\beta_q\right)$$

Formula Explanations:

(a) Scaling:$\frac{1}{s}x$ scales the input x to the quantized integer range.

(b) Adding the Zero-Point z is added to align the zero-point.

(c) Rounding: The scaled result is rounded to the nearest integer.

(d) Clip Function: The result is clipped to ensure it remains within the quantized range $[\alpha_q,\ \beta_q]$

6. **Quantized Matrix Multiplication** When both the input and weights of a matrix are quantized, the floating-point matrix multiplication needs to be transformed into quantized integer multiplication. The original floating-point matrix multiplication is:

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

In quantized form, the calculation becomes:

$$Y_{q,i,j}=z_Y+\frac{s_b}{s_Y}\left(b_{q,j}-z_b\right)+\frac{s_Xs_W}{s_Y}\left(\sum_{k=1}^{p}X_{q,i,k}W_{q,k,j}-z_W\sum_{k=1}^{p}X_{q,i,k}-z_X\sum_{k=1}^{p}W_{q,k,j}+pz_Xz_W\right)$$

This formula reflects the quantized matrix multiplication process, incorporating adjustments for inputs, weights, biases, and zero-points.

7. **Simplified Calculation Formula** When the zero-pointsz_X, z_W, and z_Y are all equal to zero, the extra terms in the formula disappear, and the matrix multiplication formula simplifies to:

$$Y_{q,i,j}=\frac{s_b}{s_Y}\left(b_{q,j}\right)+\frac{s_Xs_W}{s_Y}\left(\sum_{k=1}^{p}X_{q,i,k}W_{q,k,j}\right)$$

When the zero-points are zero, the terms involving the zero-points are removed, leaving only the scaled sums of quantized inputs, weights, and biases. This significantly reduces the computational load and is especially beneficial for low-power or resource-constrained devices.
For sake of simplification, the strategy in this article of quantization keeps symmetric.

2.3 Post-Training Quantization (PTQ)

Post-Training Quantization (PTQ) is a widely employed optimization technique designed to enhance the efficiency of neural networks after they have been fully trained. Its primary objective is to reduce computational complexity and memory consumption,by converting the model's precise floating-point weights and activations into more compact, lower-precision integer formats, such as 8-bit integers. This conversion leads to increased inference speed and reduced storage demands without necessitating model retraining, making PTQ particularly advantageous for deployment on resource-constrained devices.

Despite its benefits, PTQ presents certain limitations. The process typically relies on a small calibration dataset to approximate the activation ranges within the model. However, this dataset may not comprehensively represent the full range of potential inputs, leading to quantization errors. Such errors are especially pronounced when activations exhibit complex or highly variable distributions, potentially resulting in a decline in model accuracy when exposed to diverse or intricate data that was not adequately captured by the calibration dataset.

To mitigate these challenges, Quantization-Aware Training (QAT) is employed. QAT incorporates quantization into the training process itself, simulating low-precision arithmetic during training. This approach enables the model to adapt its weights and activations in real-time, thereby minimizing the accuracy degradation typically associated with quantization. As a result, QAT facilitates more robust model performance post-deployment, ensuring that the quantized model retains higher accuracy in real-world applications.

2.4 Quantization-Aware Training (QAT)

Quantization-Aware Training (QAT) integrates the process of quantization directly into the training of neural networks, enabling models to operate effectively in low-precision environments without significant loss of accuracy. During QAT, both weights and activations are represented in lower-precision formats, such as 8-bit integers, during the forward pass, thereby simulating the conditions the model will face in deployment. However, the backward pass, essential for the learning process, remains in floating-point precision, ensuring that the model continues to update and improve without disruption.

The primary advantage of QAT is its capacity to prepare the model for quantized inference during training, allowing it to adjust to the challenges of low-precision computation. This approach ensures that the model retains high accuracy, even when subjected to the constraints of quantized inference. Unlike Post-Training Quantization (PTQ), which depends on a separate calibration dataset to estimate activation ranges, QAT incorporates quantization into the training loop itself, making it more adaptive and effective for complex neural networks and diverse datasets.

One of the key challenges of QAT is the non-differentiability of the quantization process, which can hinder the backpropagation of gradients. To overcome this issue, the Straight-Through Estimator (STE) is employed. The STE approximates gradients during quantized operations, allowing the backpropagation process to proceed smoothly despite the non-differentiable nature of quantization.

In summary, QAT enables neural networks to efficiently adopt low-precision computation while preserving high accuracy. Although more computationally demanding than PTQ, QAT produces superior results by generating models that are both efficient and highly optimized, making them well-suited for demanding tasks such as image classification and speech recognition, with minimal performance degradation.

2.5 Quantization tool-Qkeras

QKeras, an open-source extension of Keras, has emerged as a powerful framework in deep learning, where the balance between efficiency and accuracy is critical. With precision and ingenuity, QKeras facilitates the transformation of high-precision weights and activations, reducing them from floating-point representations to lower-precision formats. This reduction significantly decreases the computational burden while retaining the accuracy and power demanded by modern neural networks.

At the heart of QKeras are its quantized layers, each crafted with optimization as a priority. The QConv2D layer, for instance, quantizes both kernels and inputs, enabling efficient two-dimensional data processing while minimizing computational overhead. Similarly, the QDense layer applies quantization to both weights and biases, effectively reducing parameter sizes and storage needs without sacrificing model fidelity.

The QActivation layer, though often subtle in its role, is essential in applying quantization to activation functions, thereby lowering computational complexity without diminishing their functional impact. A particularly notable feature of QKeras is the fusion of QConv2D with BatchNormalization, which eliminates redundant operations and optimizes the model's efficiency. This integration enhances processing speed while ensuring smooth and high-performance calculations.

QKeras provides developers with the tools to create neural networks that are both lightweight and highly efficient, making them ideal for deployment on resource-constrained devices without a significant flop in accuracy. It embodies the balance between functionality and computational efficiency, enabling even modest hardware to meet the demands of current machine learning applications.

The cornerstone of QKeras is the quantizer, an essential component akin to a sculptor's chisel, which transforms tensors from high-precision to lower-precision representations. The quantizer serves as a critical bridge between floating-point data and reduced precision, managing computational complexity while preserving the integrity of the data.
Among the most commonly used quantizers in QKeras is the quantized_bits quantizer, which plays a pivotal role in maintaining accuracy while reducing the complexity of the model.
here are the parameters of quantized_bits quantizer

- bits: Specifies the number of bits used for quantization.
- integer: Denotes the number of bits allocated to represent the integer part of

the number.

- symmetric: Ensures that the quantization is symmetric, meaning it provides equal ranges for positive and negative values.
- alpha: A scaling factor that can be set to auto_po2 or auto, calculated per channel to fine-tune the quantization process.
- keep_negative: Determines whether negative values should be retained or clipped.
- use_stochastic_rounding: Activates stochastic rounding, which introduces controlled randomness to enhance training stability.
- scale_axis: Specifies the axis along which scaling occurs.
- qnoise_factor: Controls the level of quantization noise added to the outputs.
- use_ste: If enabled, utilizes the "straight-through estimator" (STE), commonly used for gradient estimation during backpropagation in quantized models.

I would like to focus on introducing one of the parameters, which is the alpha parameter. It is generally divided into two modes, namely auto and auto_po2. Here, I will mainly introduce the auto_po2 mode.
When alpha is set to po2,the scale factor is constrained to powers of two, which fundamentally impacts the subsequent computational operations.
When alpha is set to po2, the complexity of multiplication is considerably reduced, as multiplications are replaced by more efficient bit-shifting operations. from the equation mentioned before,the multiplications impacted by the scale factors a lot.

$$Y_{q,i,j} = \frac{s_b}{s_Y}\left(b_{q,j}\right) + \frac{s_X s_W}{s_Y}\left(\sum_{k=1}^{p} X_{q,i,k} W_{q,k,j}\right)$$

This transformation eliminates the computational burden traditionally associated with multiplications, enabling hardware to execute operations more efficiently. The use of bit-shifts allows data processing to occur at a faster rate, which is particularly advantageous in resource-constrained environments where minimizing computational overhead is crucial.
However, as with any optimization, setting the scaling factor to powers of two—such as in the case of auto_po2—introduces certain limitations. While this configuration enhances computational efficiency, it narrows the range of available scaling factors. This restriction can create a trade-off: although operations become more efficient, the model's ability to finely adjust its parameters in response to complex data may be compromised. The implications of this trade-off, though subtle, can affect the network's performance over time. These potential drawbacks, emerging from the pursuit of efficiency, warrant further examination as we delve deeper into the discussion.

2.6 Mixed precision Quantization and Autoqkeras

2.6.1 Mixed precision Quantization

Mixed Precision Quantization (MPQ) is an advanced technique employed to reduce the computational complexity of deep neural networks (DNNs) by applying varying levels of numerical precision to different layers or operations within the network. Instead of uniformly using high precision, such as 32-bit or 16-bit floating-point values across the entire model, MPQ strategically assigns lower precision (e.g., 8-bit or lower) to layers where it has minimal impact on performance, while maintaining higher precision where necessary to preserve model accuracy.

The primary objective of MPQ is to optimize the balance between model accuracy and computational efficiency. By selectively reducing precision, MPQ minimizes memory usage and accelerates inference, making it particularly suitable for resource-constrained environments such as edge devices and embedded systems.

2.6.2 Autoqkeras

AutoQKeras builds on the foundation of QKeras by providing an automated framework specifically designed to optimize quantization strategies for deep learning models. It is especially well-suited for resource-constrained environments, where the balance between performance and efficiency is crucial. With AutoQKeras, there is no need for manual tuning of bit-widths or quantization schemes across layers—this process is entirely automated, minimizing precision loss while effectively managing computational demands.

At its core, AutoQKeras focuses on optimizing quantizers throughout the network, a critical task for models deployed on hardware with limited computational capacity. It achieves this by thoroughly analyzing the sensitivity of each layer to quantization while accounting for hardware constraints. AutoQKeras employs advanced techniques such as Bayesian optimization, systematically exploring various configurations to identify the optimal setup for each model.

A standout feature of AutoQKeras is its use of mixed-precision quantization. This approach adjusts the bit-widths of layers based on their significance to overall model accuracy. Layers that are essential for maintaining high accuracy are allocated higher precision, while less critical layers are more aggressively quantized. This method enhances processing speed and reduces memory consumption, all while preserving accuracy in the most important areas of the model.

AutoQKeras is also adaptable to specific hardware environments, allowing users to impose constraints such as limiting bit-widths or adhering to power consumption targets. This flexibility makes it particularly advantageous for edge devices and hardware accelerators, enabling efficient deployment without sacrificing performance.

The process begins by defining a neural network using QKeras layers, which already support quantization. AutoQKeras then automates the search for the optimal quantization configuration through iterative training and testing. Once the ideal configuration is identified, the model is fine-tuned with the optimized quantization parameters, preparing it for deployment on the target hardware.

In AutoQKeras, the key to optimization lies in selecting the most appropriate quantizers for each layer and determining their optimal bit-widths. While reducing bit-width can impact accuracy, AutoQKeras avoids focusing solely on precision. Instead, it seeks a balance where model performance and resource efficiency coexist, ensuring that the final model is both accurate and computationally efficient.

To achieve this balance, AutoQKeras introduces the Forgiving Factor (FF), a tool designed to manage the trade-off between cost of calculation and accuracy. The FF acts as a guiding parameter, ensuring that the algorithm considers not only the finest precision but also the efficiency of computational load. By carefully calculating the Forgiving Factor, AutoQKeras strikes a delicate balance between these forces, enabling models to achieve both accuracy and efficiency in their final form.

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

this is the formula of FF [22]
In this equation, rate and stress are kept constant, while Delta changes depending on the result of the trial model. It is set to delta_p if the trial model's cost is lower than the reference model's cost, and delta_n if it is higher. The model's cost can be measured by either bit-width or energy use, but because energy depends on many hardware factors, this explanation focuses on optimizing bit-width.

AutoQKeras, aiming for balance, calculates the trial model's cost by tracking bit usage over several iterations. The reference cost comes from a model with fixed bit-widths, providing a baseline for comparison. By using the Forgiving Factor (FF), AutoQKeras ensures that accuracy is preserved while Keras Tuner looks for smaller and more efficient networks. The final evaluation score is calculated using a straightforward formula that balances efficiency and performance.

The final score used for evaluation is :

$$\text{score} = \text{accuracy} \times \text{FF}$$

Additionally, AutoQKeras includes an optional feature that adjusts filter sizes to help maintain performance when lower bit-widths are applied during quantization. However, in this study, filter tuning remains inactive and is not modified.

By using AutoQKeras, the complex and time-consuming task of manually tuning quantization settings is replaced by an automated process that adapts easily to specific hardware constraints. The real advantage of AutoQKeras is its ability to create high-performance models optimized for environments where efficiency is essential. From embedded systems to mobile devices, where computational power is limited, AutoQKeras ensures that both efficiency and performance are optimized without reducing the model's overall effectiveness.

2.6.3 Extension of Qkeras

"When running one of these quantized kernels, QKeras partially uses the technique called "fake quantization" that is the same technique used by Tensorflow Lite. This technique consists in quantizing and dequantizing inputs and weights before running the floating-point kernel. In this way, inputs, weights (and then outputs) remain floating point numbers, but can represent quantized values only. However, there is a difference: QKeras fake-quantizes only weights and biases and does not fake-quantize the inputs, i.e. they remain "true" floating point numbers so they can represent any number in the floating point range The same holds also for the outputs: they remain in floating point because computing a kernel with floating-point inputs and fake-quantized weights gives floating-point outputs
To tackle this problem, we create a new class called quantized_bits_featuremap() which is a quantizer that implements the affine quantization mapping formula."[23]

quantized_bits_featuremap was introduced as a new quantization method designed specifically to handle the inputs of different layers, filling the gap where traditional fake quantization methods lacked an input quantizer. Building on this idea, I added more features to this quantizer by adjusting the scale factor and limiting it to powers of two. This change makes the quantizer more efficient for hardware use, as it allows the system to use simple bit-shift operations instead of complex multiplications, making the computations faster and more practical. in anoter word, quantized_bits_featuremap quantizers support auto_po2 mode from now on.

Chapter 3

Quantization Benchmarking

3.1 Overview of Quantization Benchmarking

Edge computing, in contrast to cloud-based solutions, reduces latency and power consumption by processing data locally, thereby enhancing both operational efficiency and data security. This article concentrates on the development of quantized models specifically tailored for portable and embedded devices, with the goal of optimizing edge computing systems. To further minimize computational complexity during inference, the scale factors of the quantizers were constrained to powers of two, transforming multiplication operations into bitwise shifts—an approach significantly more efficient than conventional multiplications or divisions.

To accomplish this, QKeras was employed alongside an extended custom quantizer, as well as AutoQKeras, which utilizes Bayesian optimization to identify the optimal quantization configurations for weights, biases, and activation functions across all layers of the model. The primary aim was to implement lower-bit quantizers throughout the model while preserving accuracy.

Six distinct models were evaluated in this study. MobileNetV1, MobileNetV2, and EfficientNet-B0 were assessed on a common dataset due to their similar architectural underpinnings. Additionally, an autoencoder, a conventional CNN, and ResNet were trained and tested on three separate datasets. The detailed performance of these models will be presented in the subsequent sections, with a particular emphasis on their quantization efficacy and resource efficiency in edge computing environments.

3.2 Framework and Execution

3.2.1 Project Structure and Objectives

The primary motivation for this project stems from the increasing demand for efficient machine learning models capable of operating on low-power, resource-constrained devices used in edge computing. As highlighted, edge computing offers key advantages over cloud computing, including reduced latency, lower power consumption, and enhanced security through local data processing. This research focuses on developing quantized deep learning models specifically optimized for mobile and embedded systems, with particular emphasis on Mixed-Precision Quantization (MPQ). The objective is to explore how MPQ can be tailored to meet the performance and efficiency requirements of edge devices.

3.2.2 Research Methodology and Technical Details

In this study, six models were trained and evaluated to assess their accuracy, efficiency, and overall performance, with a special focus on the impact of MPQ. The research aims to analyze the performance improvements enabled by MPQ, while identifying challenges and proposing solutions to overcome them. By quantizing the models and employing methods such as scaling factors restricted to powers of two, the computational load during inference is reduced, decreasing both memory usage and computational complexity.

3.2.3 Experimental Designs

The models utilized in this research include MobileNetV1, ResNet, an autoencoder, and a classic CNN. These models were previously optimized for MPQ under the alpha=auto setting in earlier studies, and this work leverages those results, eliminating the need for a lengthy optimization process. The focus of this project is to explore the use of alpha=auto_po2, retraining and fine-tuning the pre-configured models to evaluate the effectiveness of this setting. Additionally, a comparative analysis is conducted by applying flat quantization and benchmarking the outcomes against MPQ with 16-bit, 8-bit, and 4-bit quantization configurations. This analysis aims to provide insights into the trade-offs between model size, accuracy, and computational efficiency.

3.2.4 Results Analysis and Discussion

The results of the experiments, including a detailed analysis of model performance across different quantization settings, will be discussed in the following sections. The research also highlights challenges faced during the study and presents potential future directions, such as exploring novel optimization techniques and refining quantization strategies for edge computing applications. The primary objective of this project is to demonstrate how MPQ, particularly using the auto_po2 configuration, can be applied to create efficient, quantized models suitable for edge computing. By thoroughly evaluating these models, the research seeks to offer valuable insights into the practical implementation of quantized deep learning models on embedded systems, contributing to the advancement of efficient machine learning techniques for resource-limited environments.

3.3 project outline

3.3.1 System Configuration

Software and Hardware Provided by Politecnico di Torino Laboratory
The software and hardware resources used in this project were provided by the Politecnico di Torino laboratory. The specific configurations are as follows:
Computing Platform:cuda:10.2
Processor:Intel(R) Core(TM) i5-7400 CPU @ 3.00GHz
GPU:Nvidia GTX 1070
Memory: 16G
Storage: 4T
Operating System: CentOS 7
Python 3.8.13
TensorFlow 2.4.0
Keras 2.11
QKeras 0.9.0 custom
Keras-Tuner 1.3.0
TensorBoard 2.10.1
Librosa 0.9.2
PyDub 0.25.1
FFprobe 0.5
Jupyter 1.0.0
By clearly specifying the hardware and software configurations provided by the laboratory, this section helps establish the technical foundation for the project and allows for reproducibility in future research.

3.3.2 Project Organization and Directory Overview

- **scripts**:This is the top-level directory, primarily responsible for preprocessing datasets and controlling different functionalities. It serves as a master switch for various tasks, including AutoQKeras search, flat quantization with uniform quantization, or fine-tuning and testing the models.

- **dataset**: Contains the datasets used for model training, as well as the training sets specifically for quantized versions of the models.

- **ref_model**: This directory holds models derived from reading related literature or obtaining source code from GitHub. These models are used for standard floating-point training, allowing for comparison with the quantized versions.

- **model_mod**: Compared to ref_model, this directory includes models with a special activation quantization layer (Sigmoid). The models are directly quantized using QKeras, where the Sigmoid layers are replaced with the custom quantized_bit_featuremap layer mentioned earlier

- **best_model**: Stores the best MPQ model configurations identified through AutoQKeras, used for fine-tuning.

- **Others**: This includes various tools for analysis, such as H5 file processors, tools for analyzing zero-value ratios in the trained weights and biases, and plotting scripts to facilitate research on encountered issues.

- **results**: Contains the results obtained from the models mentioned above or from other analytical tools used in the project.

3.3.3 Overview of the Training Process

1. **Loading pretrained weights** First, the weights from the normally trained model are loaded into a model that has been modified with special processing layers. This allows the model to converge within a few epochs during training.

2. **Setting Bayesian Search Parameter** The parameters for Bayesian optimization are configured, defining the quantization limits, the maximum number of trials, weight transfer, and most importantly, the selectable range of quantizers. For example, the kernel and activation are assigned 4, 8, or 16 bits as options.
   For kernels:
   quantized_bits(4,4,1,1,alpha='auto'): 4,
   quantized_bits(8,8,1,1,alpha='auto'): 8,
   quantized_bits(16,16,1,1,alpha='auto'): 16
   For activations:
   quantized_bits_featuremap(4,4,1,1,alpha='auto_po2',scale_axis=0): 4,
   quantized_bits_featuremap(8,8,1,1,alpha='auto_po2',scale_axis=0): 8,
   quantized_bits_featuremap(16,16,1,1,alpha='auto_po2',scale_axis=0): 16
   while bias is assigned 16 or 31 bits.
   quantized_bits(16,16,1,1,alpha='auto_po2'): 16,
   quantized_bits(31,31,1,1,alpha='auto_po2'): 31
   The total number of combinations is dependent on the number of layers that require quantization. For instance, in the case of MobileNetV1, there are 28 kernel weights, 28 bias weights, and 36 activations that need quantizers, resulting in 508,032 possible combinations (28 x 3 x 28 x 2 x 36 x 3). Using grid search or hyperband methods would be computationally expensive and yield sub-optimal results. Therefore, Bayesian optimization is the best solution here.
   These parameter configurations are set within Python dictionaries, and the built-in functions in AutoQKeras will read these settings through run_config.

3. **Setting Callback Functions** Callback functions are executed at the end of each epoch to perform various actions on the training results. In this experiment, the key callback functions include:

   - **Early Stopping**: Stops the training of underperforming models
   - **CSV Logger**: Logs the loss, accuracy, validation loss, validation accuracy, and other metrics for each trial.
   - **Checkpoint**: Saves the best model weights at the end of each epoch.
   - **Tnsor-board Support**: Generates TensorBoard plot
   - **lrs_callback** :sets the leanring rates upon different epochs

4. **Launching AutoQKera** The code is executed as described above, and AutoQKeras is launched.

5. **Selecting the Best Model** AutoQKeras will automatically select the best model based on the scoring formula mentioned earlier.

6. **Training the Best Model** The best model selected in the previous step is re-trained according to the dataset, with the quantizers' auto mode being switched to auto_po2.

7. **Model Evaluation** Finally, the most recent training results are evaluated to assess the model's performance.

3.4 Anomaly Detection

Overview and model:
The first benchmark is Anomaly Detection,before hop into it, the model so called fc auto encoder should be introduced at first
The FC-AutoEncoder, a sophisticated model in the field of machine learning, is built on fully connected layers and excels in tasks like dimensionality reduction and feature extraction. Its encoder works like a careful sculptor, trimming away unnecessary details to distill the data into a compact representation in the latent space. The decoder, in turn, reconstructs the original input with remarkable fidelity. Though it follows the simple structure of a symmetric multilayer perceptron, the FC-AutoEncoder possesses a certain elegance, revealing the essential patterns within the data, much like a well-crafted narrative unveils hidden truths.

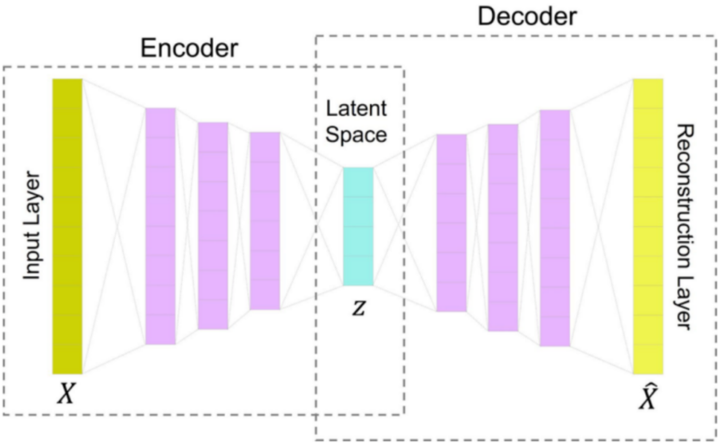


Figure 3.1: illustration of autoencoder.[26]

The dataset is composed of ToyADMOS and MIMII[24],featuring audio recordings from six distinct types of machines: fan, valve, pump, toy conveyor, and toy car. For each machine type, there are four individual machines, each identified by a unique ID. Every ID contains over 1000 samples, with each sample being a 10-second recording that includes background noise.
The dataset consists of a development dataset, an additional training dataset[25], and an evaluation dataset. The training data is unlabeled, while only the test set bears the marks of whether the machine operates in normalcy or anomaly. The additional training dataset contains solely unlabeled normal data, with a total of 3000 samples. As for the evaluation dataset, it shall not be employed in this experiment.
During the training process, the objective is to optimize the model's weights so that the audio data can be reconstructed with minimal error, thereby ensuring that the

output closely approximates the input. To achieve this, training is conducted using the training sets from both the development dataset and the supplementary training dataset, resulting in a total of four distinct training sets. Upon completion of the training phase, the model is evaluated on the final test set.

For each data instance in the test set, the model's output is compared to the original input to compute an "anomaly score." Utilizing the true labels of the test data—whether classified as normal or anomalous—alongside the corresponding anomaly scores, a Receiver Operating Characteristic (ROC) curve is plotted. This curve illustrates the model's ability to discriminate between normal and anomalous instances, providing a comprehensive assessment of its performance.

$$\text{error} = \text{MSE}(\text{input} - \text{prediction})$$

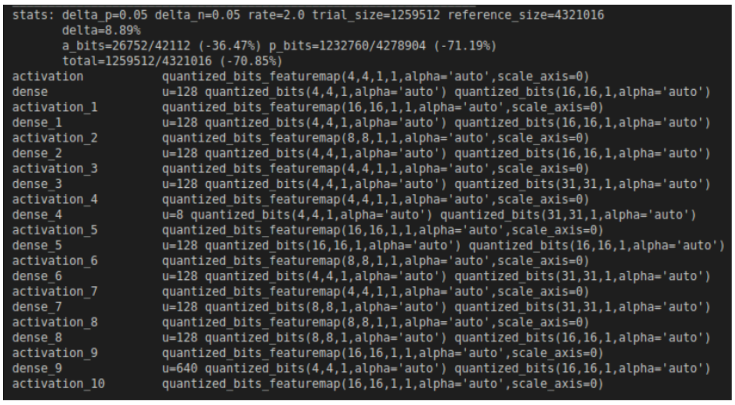
$$\text{anomaly\_score} = \frac{1}{n}\sum_{i=1}^{n} \text{error}_i$$

$$\text{AUC} = f(y_{\text{true}}, \text{anomaly\_score})$$

However, for AutoQKeras, a metric with a value between 0 and 1 is necessary to enable the calculation of its Forgiving Factor. Clearly, Mean Squared Error (MSE) does not satisfy AutoQKeras' requirements. To resolve this, we developed a novel custom metric specifically designed to meet the criteria of AutoQKeras:

$$\text{custom\_metric} = \frac{1}{1 + \text{loss}^{10}}$$

his adjustment aligns the metric's nature with that of accuracy, making it compatible with AutoQKeras' optimization process.
The best mixed-precision configuration of FC-AutoEncoder has been found by previous researcher.[25]



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

Figure 3.2: Best mixed-precision FC-AutoEncoder configuration found by AutoQKeras. [25]

In the case of this project, all alpha =auto will be replaced into auto_po2
The training parameter is set as:
Learning rate schedule:

- Training/Validation split: 90
- Batch size: 100
- Epochs: 100
- Optimizer: Adam optimizer for model training
- Loss function: MSE
- Metrics tracked: Accuracy
- Input handling: Data is shuffled during training
- Alpha=auto_po2

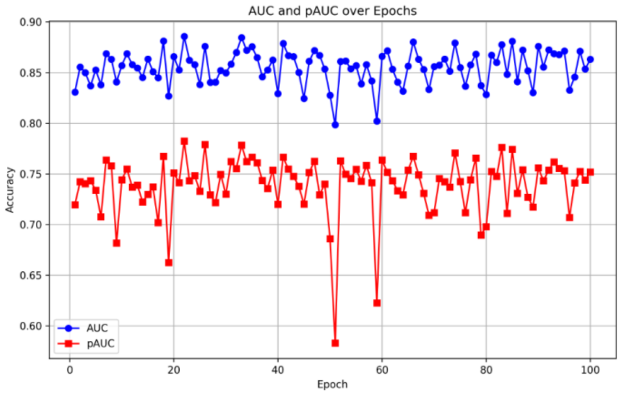


Figure 3.3: results of MPQ autoencoder over 100 epochs

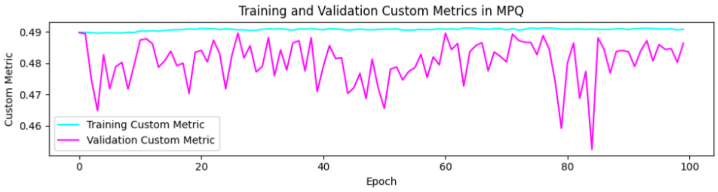


Figure 3.4: costumed metrics values over 100 epochs

From the graph, it can be observed that the AUC curve (in blue) remains relatively stable, with a slight upward trend, particularly in the later stages of training, stabilizing between 0.83 and 0.88. This suggests that the overall classification performance of the model is strong and demonstrates improvement over time. In contrast, the pAUC curve (in red) displays significant fluctuations, particularly around the 60th epoch, where a notable decline occurs. Since pAUC is calculated within the region where the False Positive Rate (FPR) is less than 0.1, this indicates that the model's performance in the low FPR region is less stable compared to its overall performance. This instability in handling critical areas highlights potential weaknesses. Therefore, while the model's overall classification performance is satisfactory, further optimization is necessary to ensure more reliable performance in critical regions with low FPR, especially in high-risk scenarios.

3.5 Key words spotting

In this keyword classification task, we use a Depthwise Separable Convolutional Neural Network (DS-CNN) to classify audio signals that contain specific keywords. The dataset is sourced from Speech Commands V2 [27] and contains 30 different words. Of these, 10 are the target keywords, while the remaining 20 are non-keywords. Each word has about 3,530 recordings, contributed by 2,618 speakers from different regions and countries, providing a range of accents. For this classification task, we focus on 12 categories: 10 for the target keywords like "down," "go," and "yes," along with two extra categories, "silence" and "unknown." The "silence" category includes recordings with no speech or only background noise, while the "unknown" category includes all other words outside of the 10 target keywords.

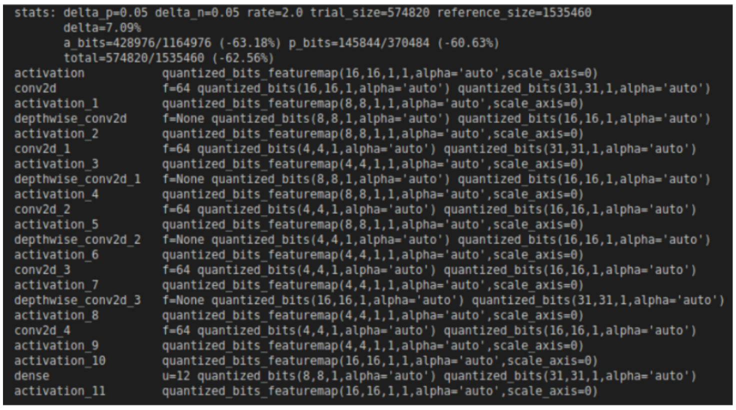
To better simulate real-world conditions, the recordings have an 80% chance of being mixed with background noise, where the noise level randomly varies between 0 and 0.1 of the speech volume. Since the raw audio files cannot be directly used for training, we use a technique called Mel-Frequency Cepstral Coefficients (MFCC) [28] to convert the audio signals into numerical features. MFCC is commonly used in tasks like speech recognition and keyword spotting because it extracts the essential characteristics of audio, making it easier for machines to process and understand speech. Essentially, MFCC acts as a "translator," turning complex sound waves into simpler numerical representations.

What makes MFCC particularly useful is that it focuses on how humans perceive sound. As listeners, we are more sensitive to lower frequencies (like deep sounds) and less sensitive to higher frequencies (like sharp tones). MFCC uses the Mel scale to emphasize the frequencies our ears are naturally attuned to. Through several mathematical transformations, MFCC converts complex frequency data into coefficients that capture the key elements of the audio, which is critical for tasks like speech recognition and keyword detection. For example, in voice assistants like Siri or Alexa, MFCC helps extract features from the user's speech, allowing the system to accurately recognize spoken commands.
Once the data is processed, we proceed to train the model. In the auto_po2 quantization mode, Mixed-Precision Quantization (MPQ) demonstrates exceptional performance, coming very close to the accuracy achieved by traditional floating-point models. The charts below will offer a more detailed analysis of these results, offering meaningful insights into the model's behavior under different configurations.
The training parameter is set as

- Learning rate schedule:0.00025
- Training/Validation split: 90
- Batch size: 100

- Epochs: 70 in MPQ, 50 in flat
- Optimizer: Adam optimizer for model training
- Loss function: Sparse Categorical Crossentropy
- Metrics tracked: Accuracy
- Input handling: Data is shuffled during training
- Alpha=auto_po2

In previous research, the optimal Mixed-Precision Quantization (MPQ) configuration has already been identified through AutoQKeras.[25] Building on this foundation, I will use their results and modify the model to allow it to also run in auto_po2 mode. This adjustment aims to further optimize the model's performance while exploring the effects in different quantization modes.



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

Figure 3.5: Best mixed-precision configuration for key word spotting model found by AutoQKeras. [25]

the results display below:

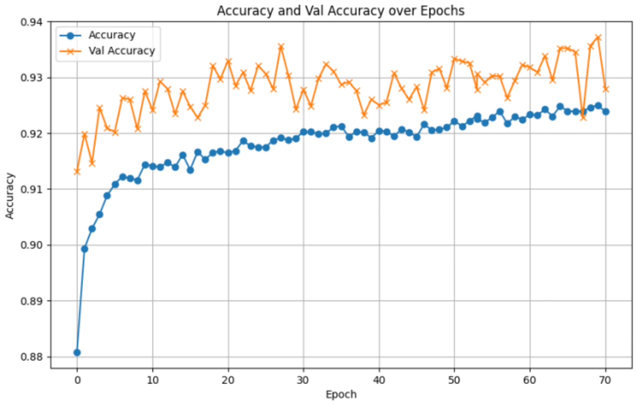


Figure 3.6: accuracy of kws model over epochs

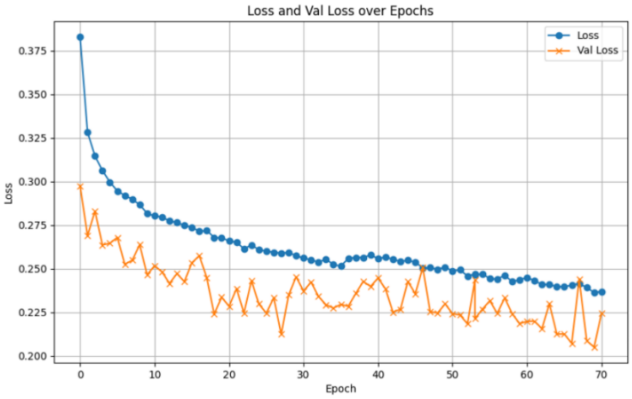


Figure 3.7: loss of kws model over epochs

The model demonstrates a relatively stable performance during training. Training accuracy steadily increases and eventually plateaus, while validation accuracy, though exhibiting some fluctuations, remains consistently high, indicating that the model has strong generalization capabilities. Training loss shows a continuous downward trend, and despite occasional oscillations, validation loss also declines, suggesting that the model performs well on both the training and validation sets. Overall, the model's performance is stable, with the trends in accuracy and loss behaving predictably—similar to a diligent student who may occasionally present an unexpected result but generally performs reliably.
Here are the flat bits 4, 8 and 16bits

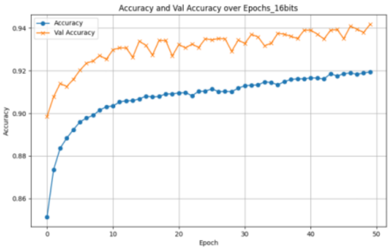


Figure 3.8: Accuracy over Epochs in 16 bits

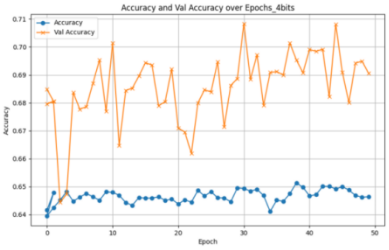


Figure 3.9: Accuracy over Epochs in 4 bits

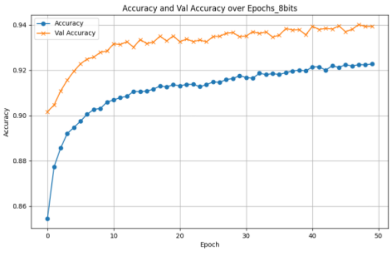


Figure 3.10: Accuracy over Epochs in 8 bits

In the keyword recognition task, comparing different quantization precisions reveals that 4-bit quantization results in significant instability, with low training accuracy and highly fluctuating validation accuracy, making it difficult to ensure the model's generalization capability. Under 8-bit quantization, the model's performance improves significantly, with both training and validation accuracy stabilizing, and validation accuracy remaining at a high level, demonstrating strong generalization ability. In contrast, 16-bit quantization does not provide notable additional performance benefits, indicating that 8-bit quantization already offers sufficient precision to ensure good model performance. Overall, the 8-bit quantization mode strikes the best balance between model performance and computational resources, particularly when combined with MFCC feature extraction and the auto_po2 quantization mode, achieving optimal results in both accuracy and computational efficiency.

3.6 Image classification

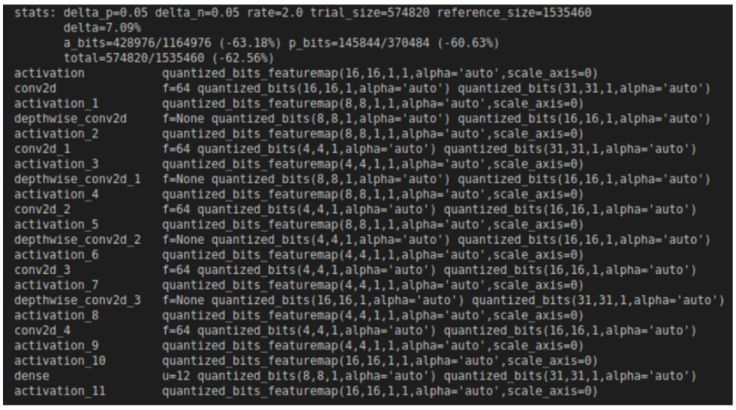
This task primarily involves training a model in machine learning to classify various objects in images, using the ResNet neural network. It has widespread applications in real-world scenarios, such as in the field of autonomous driving. In this task, the CIFAR-10 dataset was used, which was compiled by the machine learning research group at the University of Montreal in Canada. This dataset contains 60,000 color images, each with a size of 32x32. These images are meticulously categorized into 10 classes: airplanes, automobiles, birds, cats, deer, dogs, frogs, horses, ships, and trucks.[34] Each category is exclusive, with 6,000 images representing the unique characteristics of their respective classes. The dataset is extensively utilized in machine learning and deep learning for tasks such as image classification and object detection, supporting both training and evaluation of models.

To ensure efficient file access, the images are converted into a binary format using pickle, which boosts the reading speed. The dataset is split into 5 batches, each containing 10,000 images, while an additional batch with 10,000 images is set aside as the test set. Additionally, there is an option to use a lighter test set containing only 200 images for performance evaluation, offering different levels of testing.

The training parameters are as follows:

- Learning rate decay formula:0.001 × 0.99êpoch
- Training/Validation split: 90
- Batch size: 30
- Epochs: 500
- Optimizer: Adam
- Loss function: categorical crossentropy
- Metrics tracked: Accuracy
- Alpha=auto_po2

In previous research, the optimal Mixed-Precision Quantization (MPQ) configuration has already been identified through AutoQKeras.[25] Building on this foundation, I will use their results and modify the model to allow it to also run in auto_po2 mode. This adjustment aims to further optimize the model's performance while exploring the effects in different quantization modes.



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

Figure 3.11: Best mixed-precision configuration for ResNets found by AutoQKeras. [25]

then, the model trained in auto_po2 mode

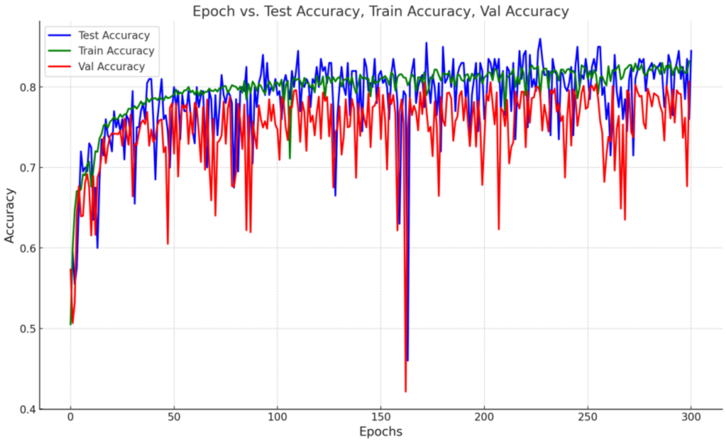


Figure 3.12: accuracy of Resnets over epochs

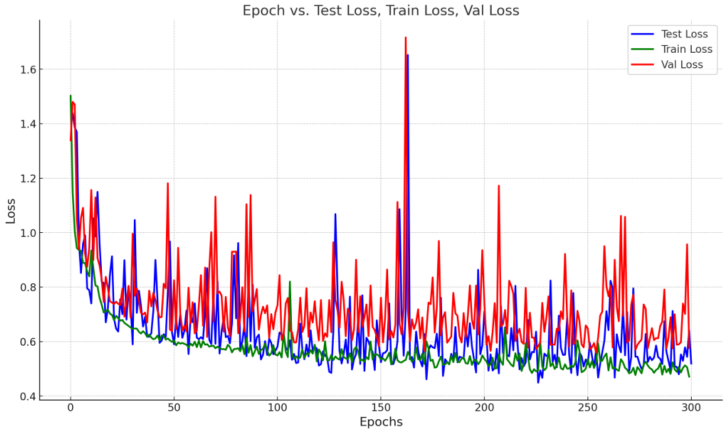


Figure 3.13: loss of Resnets over epoch

This is the result of training under mixed precision quantization mode. In the Loss Chart, the Test Loss (test set loss) is significantly higher than the Train Loss (training set) and Val Loss (validation set) in the early stages of training, indicating that the model's performance on the test set is not as good as on the training and validation sets. The Train Loss and Val Loss follow a similar trend, though the validation set loss is slightly higher at certain points, showing some signs of overfitting. As training progresses, all three losses decrease, indicating that the model is continually optimizing and gradually converging.
In the Accuracy Chart, the test set accuracy is relatively low, especially early on, significantly lower than the training and validation set accuracy. Both Train Accuracy and Val Accuracy increase markedly as training progresses, with only a small gap between them, suggesting the model performs similarly on the validation set as it does on the training set, without showing clear signs of overfitting.
Overall, as training progresses, the model's losses decrease and its accuracy improve, with consistent performance between the training and validation sets, though there remains a notable gap between the test set's performance and that of the training and validation sets.
However the results of flat quantization are not presented, it reason will be discussed later.

3.7 Visual Wake Words

This is the final task, which combines the characteristics of the previous image classification and wake word recognition tasks and is referred to as Visual Wake Words (VWW). In this experiment, we used the VWW Dataset, a binary classification dataset containing two types of images: one class labeled as "person," with 53,140 images, and the other labeled as "non-person," with 56,479 images. Each image has a size of 96×96 RGB, and they all originate from the MSCOCO2014 dataset [32]. The original images cannot be directly used and require preprocessing. In this experiment, the preprocessing tool used was buildPersonDetectionDatabase.py, which is available in the Silicon Labs GitHub repository [30].

As is evident from the dataset, this task is a binary classification problem aimed at detecting whether a person is present or absent. This technology is commonly applied in security devices such as automatic alarm surveillance systems and smart locks. However, it is important to note that the origin of the test set within the dataset is unclear [31]. There is a directory named "evaluate" in the dataset, which extracted a portion of the training/validation set for testing purposes, totaling 1,000 images. While doing this carries some risk of overfitting, the results (accuracy) are the closest to those reported in the literature [33]. If we attempt to remove this subset from the training set, the resulting model deviates from the results reported in the literature. Therefore, we decided to retain these 1,000 images in the training/validation set and continue using them as the test set in the subsequent experiments.

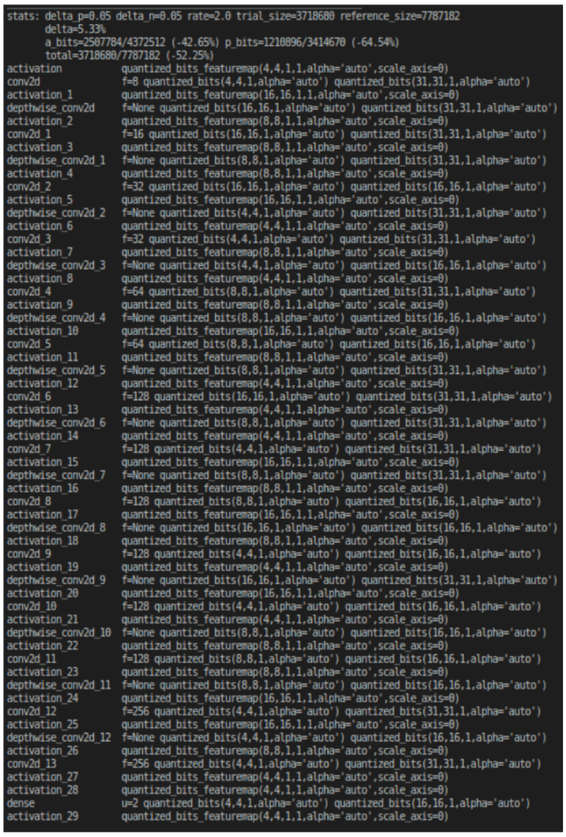
In this section, we will utilize the dataset not only for MobileNetV1, but also for MobileNetV2 and EfficientNetB0, as these networks share a similar structure.

3.7.1 Mobilenetv1

The training parameters for experiment MobileNetV1 are as follows (in a new order):

- Training/validation split: 0.1
- Training epochs: 90
- Batch size: 32
- Optimizer: Adam
- Learning rate schedule: 0.001 for the first 20 epochs, 0.0005 for the next 10 epochs, and 0.00025 until the end
- Loss function: categorical crossentropy
- Metric: accuracy
- Alpha=auto_po2

Accrording to formal student, Marco, the best configuration of MPQ s has been searched as below:



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

Figure 3.14: Best mixed-precision configuration for mobilenetv1 found by AutoQKeras. [25]

As we did above, the alpha parameters are replaced with auto_po2

The result shows as below:

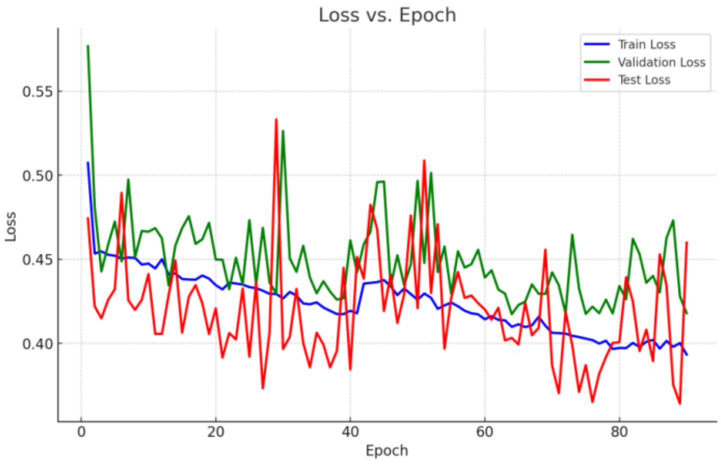


Figure 3.15: loss of MPQ mobilevnet1 over epochs

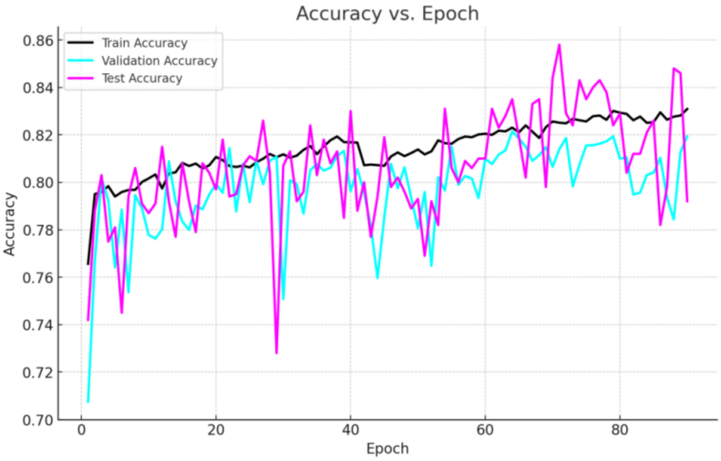


Figure 3.16: accuracy of MPQ mobilevnet1 over epochs

Under MPQ quantization, MobileNetV1 exhibited stable performance during both the training and testing phases. As illustrated in the charts, the training, validation, and test loss curves gradually decreased with increasing epochs, indicating effective convergence across various datasets. Although some fluctuations were observed in the early stages—particularly in the validation loss—all loss curves ultimately stabilized and converged.

In terms of accuracy, the training, validation, and test accuracies consistently improved as the number of epochs increased. While the validation and test accuracies were slightly lower than the training accuracy, the differences were minimal, suggesting that the quantized model retained strong generalization capabilities. Despite some initial fluctuations in test accuracy, it eventually aligned closely with

the training accuracy.
Overall, MPQ quantization significantly improved computational efficiency while having minimal impact on the model's performance, preserving MobileNetV1's accuracy and generalization. The model demonstrated robustness and stability across different datasets after quantization, highlighting the effectiveness of MPQ in maintaining performance integrity.

3.7.2 Mobilenetv2

Compared to MobileNetV1, MobileNetV2 brings forth notable refinements in its architectural design. It incorporates inverted residuals and the linear bottleneck, innovations that mitigate the potential information loss caused by traditional ReLU activations. Moreover, through its more efficient feature extraction, it elevates the model's performance, achieving greater precision and reduced computational demands, particularly in resource-constrained environments such as mobile devices. Now, let us employ the same dataset to explore the prowess of MobileNetV2.

Starting from this model, as no pre-optimized best results were provided like in the previous models, we need to begin the search from scratch. To achieve this, we will use the tool AutoQKeras, which was previously introduced in terms of its background and principles. Now, we will directly apply it. Each trial runs for 5 epochs, but if we were to run several hundred trials, the required time would be excessive. Therefore, we use a reduced version of the dataset to speed up the training process, significantly shortening the overall search time. Additionally, to ensure comparable results across different experiments, data augmentation in the typical training mode has been disabled. In total, 1000 trials were conducted, with each trial running approximately 5 epochs. The results are presented in the following plots.

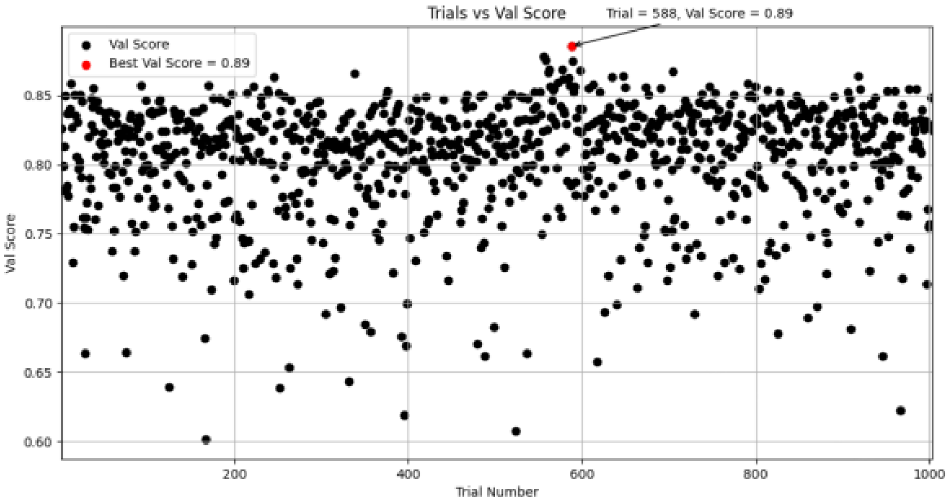


Figure 3.17: searches mbv2 scores from autoqkeras over tirals

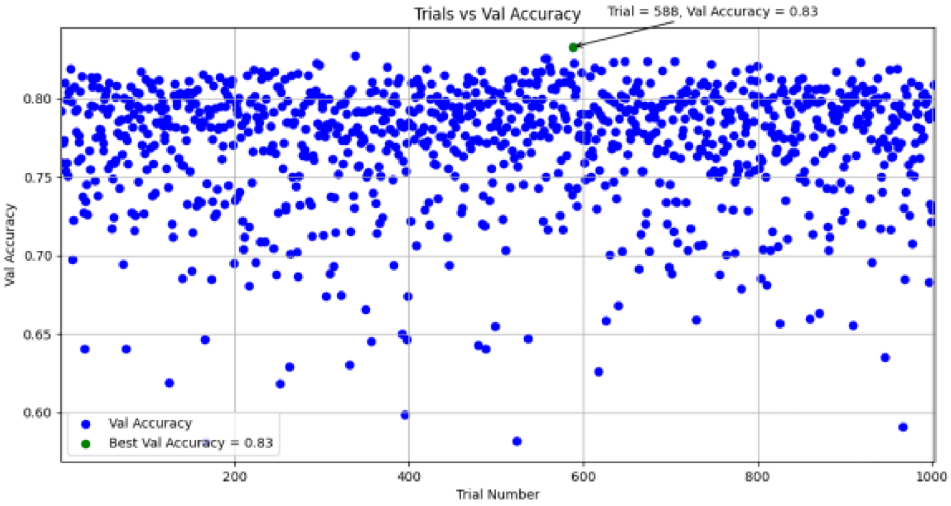


Figure 3.18: searches mbv2 accuracy from autoqkeras over tirals

These two charts show the relationship between the number of trials and validation accuracy as well as validation score. The distribution patterns of both are similar, with most validation scores and accuracy values concentrated in the higher range (around 0.75 to 0.83). trial number 588 performed the best in both charts, achieving a validation accuracy of 0.83 and a validation score of 0.89. Overall, the model's performance is relatively stable, though a few experiments yielded lower results (below 0.65), indicating the need for further analysis and optimization.
The best model, trial number 588, configured as:

```
stats: delta_p=0.05 delta_n=0.05 rate=2.0 trial_size=24252296 reference_size=58384302
       delta=6.34%
       a_bits=12023328/22761008 (-47.18%) p_bits=12228968/35623294 (-65.67%)
       total=24252296/58384302 (-58.46%)
activation           quantized_bits_featuremap(16,16,1,1,alpha=’auto’,scale_axis=0)
conv2d               f=32 quantized_bits(8,8,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_1         quantized_bits_featuremap(16,16,1,1,alpha=’auto’,scale_axis=0)
conv2d_1             f=32 quantized_bits(8,8,1,alpha=’auto’) quantized_bits(16,16,1,alpha=’auto’)
activation_2         quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
depthwise_conv2d     f=None quantized_bits(16,16,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_3         quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
conv2d_2             f=16 quantized_bits(8,8,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_4         quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
conv2d_3             f=96 quantized_bits(8,8,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_5         quantized_bits_featuremap(4,4,1,1,alpha=’auto’,scale_axis=0)
depthwise_conv2d_1   f=None quantized_bits(8,8,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_6         quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
conv2d_4             f=24 quantized_bits(8,8,1,alpha=’auto’) quantized_bits(16,16,1,alpha=’auto’)
activation_7         quantized_bits_featuremap(16,16,1,1,alpha=’auto’,scale_axis=0)
conv2d_5             f=144 quantized_bits(4,4,1,alpha=’auto’) quantized_bits(16,16,1,alpha=’auto’)
activation_8         quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
depthwise_conv2d_2   f=None quantized_bits(8,8,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_9         quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
conv2d_6             f=24 quantized_bits(16,16,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_11        quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
activation_10        quantized_bits_featuremap(4,4,1,1,alpha=’auto’,scale_axis=0)
activation_12        quantized_bits_featuremap(16,16,1,1,alpha=’auto’,scale_axis=0)
conv2d_7             f=144 quantized_bits(16,16,1,alpha=’auto’) quantized_bits(16,16,1,alpha=’auto’)
activation_13        quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
depthwise_conv2d_3   f=None quantized_bits(8,8,1,alpha=’auto’) quantized_bits(16,16,1,alpha=’auto’)
activation_14        quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
conv2d_8             f=32 quantized_bits(8,8,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
```

```
activation_15       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
conv2d_9            f=192 quantized_bits(8,8,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_16       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
depthwise_conv2d_4  f=None quantized_bits(8,8,1,alpha=’auto’) quantized_bits(16,16,1,alpha=’auto’)
activation_17       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
conv2d_10           f=32 quantized_bits(16,16,1,alpha=’auto’) quantized_bits(16,16,1,alpha=’auto’)
activation_18       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
activation_19       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
activation_20       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
conv2d_11           f=192 quantized_bits(8,8,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_21       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
depthwise_conv2d_5  f=None quantized_bits(8,8,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_22       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
conv2d_12           f=32 quantized_bits(8,8,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_23       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
activation_24       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
activation_25       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
conv2d_13           f=192 quantized_bits(8,8,1,alpha=’auto’) quantized_bits(16,16,1,alpha=’auto’)
activation_26       quantized_bits_featuremap(16,16,1,1,alpha=’auto’,scale_axis=0)
depthwise_conv2d_6  f=None quantized_bits(8,8,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_27       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
conv2d_14           f=64 quantized_bits(8,8,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_28       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
conv2d_15           f=384 quantized_bits(8,8,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_29       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
depthwise_conv2d_7  f=None quantized_bits(8,8,1,alpha=’auto’) quantized_bits(16,16,1,alpha=’auto’)
activation_30       quantized_bits_featuremap(4,4,1,1,alpha=’auto’,scale_axis=0)
conv2d_16           f=64 quantized_bits(4,4,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_31       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
activation_32       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
activation_33       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
conv2d_17           f=384 quantized_bits(8,8,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_34       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
depthwise_conv2d_8  f=None quantized_bits(16,16,1,alpha=’auto’) quantized_bits(16,16,1,alpha=’auto’)
activation_35       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
conv2d_18           f=64 quantized_bits(8,8,1,alpha=’auto’) quantized_bits(16,16,1,alpha=’auto’)
activation_37       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
activation_36       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
activation_38       quantized_bits_featuremap(4,4,1,1,alpha=’auto’,scale_axis=0)
conv2d_19           f=384 quantized_bits(4,4,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_39       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
depthwise_conv2d_9  f=None quantized_bits(8,8,1,alpha=’auto’) quantized_bits(16,16,1,alpha=’auto’)
activation_40       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
conv2d_20           f=64 quantized_bits(8,8,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_42       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
activation_41       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
activation_43       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
conv2d_21           f=384 quantized_bits(4,4,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_44       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
depthwise_conv2d_10 f=None quantized_bits(8,8,1,alpha=’auto’) quantized_bits(16,16,1,alpha=’auto’)
activation_45       quantized_bits_featuremap(16,16,1,1,alpha=’auto’,scale_axis=0)
conv2d_22           f=96 quantized_bits(4,4,1,alpha=’auto’) quantized_bits(16,16,1,alpha=’auto’)
activation_46       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
conv2d_23           f=576 quantized_bits(4,4,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_47       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
depthwise_conv2d_11 f=None quantized_bits(16,16,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_48       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
conv2d_24           f=96 quantized_bits(8,8,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_50       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
activation_49       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
activation_51       quantized_bits_featuremap(16,16,1,1,alpha=’auto’,scale_axis=0)
conv2d_25           f=576 quantized_bits(8,8,1,alpha=’auto’) quantized_bits(16,16,1,alpha=’auto’)
activation_52       quantized_bits_featuremap(4,4,1,1,alpha=’auto’,scale_axis=0)
depthwise_conv2d_12 f=None quantized_bits(8,8,1,alpha=’auto’) quantized_bits(16,16,1,alpha=’auto’)
activation_53       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
conv2d_26           f=96 quantized_bits(8,8,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_55       quantized_bits_featuremap(16,16,1,1,alpha=’auto’,scale_axis=0)
activation_54       quantized_bits_featuremap(16,16,1,1,alpha=’auto’,scale_axis=0)
activation_56       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
conv2d_27           f=576 quantized_bits(8,8,1,alpha=’auto’) quantized_bits(16,16,1,alpha=’auto’)
activation_57       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
depthwise_conv2d_13 f=None quantized_bits(8,8,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_58       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
conv2d_28           f=160 quantized_bits(8,8,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_59       quantized_bits_featuremap(16,16,1,1,alpha=’auto’,scale_axis=0)
conv2d_29           f=960 quantized_bits(4,4,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_60       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
depthwise_conv2d_14 f=None quantized_bits(16,16,1,alpha=’auto’) quantized_bits(16,16,1,alpha=’auto’)
activation_61       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
conv2d_30           f=160 quantized_bits(4,4,1,alpha=’auto’) quantized_bits(16,16,1,alpha=’auto’)
activation_62       quantized_bits_featuremap(16,16,1,1,alpha=’auto’,scale_axis=0)
activation_63       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
activation_64       quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
```

```
conv2d_31            f=960 quantized_bits(4,4,1,alpha=’auto’) quantized_bits(16,16,1,alpha=’auto’)
activation_65        quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
depthwise_conv2d_15  f=None quantized_bits(4,4,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_66        quantized_bits_featuremap(16,16,1,1,alpha=’auto’,scale_axis=0)
conv2d_32            f=160 quantized_bits(8,8,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_67        quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
activation_68        quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
activation_69        quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
conv2d_33            f=960 quantized_bits(4,4,1,alpha=’auto’) quantized_bits(16,16,1,alpha=’auto’)
activation_70        quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
depthwise_conv2d_16  f=None quantized_bits(16,16,1,alpha=’auto’) quantized_bits(16,16,1,alpha=’auto’)
activation_71        quantized_bits_featuremap(4,4,1,1,alpha=’auto’,scale_axis=0)
conv2d_34            f=320 quantized_bits(4,4,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_72        quantized_bits_featuremap(16,16,1,1,alpha=’auto’,scale_axis=0)
conv2d_35            f=1280 quantized_bits(4,4,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_73        quantized_bits_featuremap(16,16,1,1,alpha=’auto’,scale_axis=0)
activation_74        quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
conv2d_36            f=2 quantized_bits(8,8,1,alpha=’auto’) quantized_bits(16,16,1,alpha=’auto’)
activation_75        quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
```

Note: In the search phase, the auto mode was used instead of auto_po2. The auto_po2 mode is not fully refined yet, and to avoid potential instability and errors, auto mode was preferred.
The training parameters for experiment MobileNetV2 are as follows:

- Training/validation split: 0.1
- Training epochs: 100
- Batch size: 32
- Optimizer: Adam
- Learning rate schedule: 0.001 for the first 20 epochs, 0.0005 for the next 10 epochs, and 0.00025 until the end
- Loss function: categorical crossentropy
- Metric: accuracy
- Alpha=auto_po2

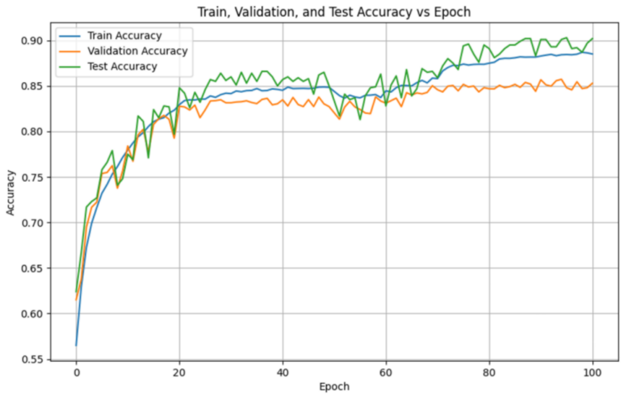


Figure 3.19: accuracy of mobilenetv2 over epochs

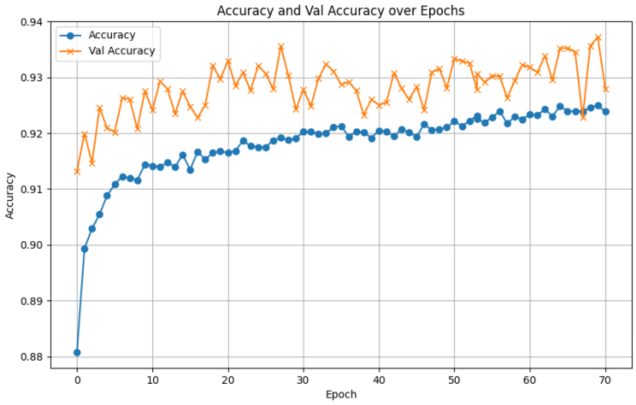


Figure 3.20: loss of mobilenetv2 over epochs

These two charts display the model's training and testing performance. In the accuracy chart, both the training and test accuracy show a clear upward trend, stabilizing towards the later stages of training. Although the validation accuracy is slightly lower, it remains relatively stable. The test accuracy exhibits some fluctuation in the early and middle stages, suggesting that there might have been some disturbances during the training process, while the training accuracy remains more consistent. Overall, accuracy steadily improves as training progresses, and around 60 epochs, the model shows signs of convergence, especially with the gap between training and test accuracy narrowing, indicating good generalization ability.

In the loss chart, the loss values for both the training and test sets decrease steadily from higher values, while the validation loss also decreases but levels off in the later stages, remaining higher than the training and test losses. Notably, the test loss falls below the training loss in the later stages, suggesting that the model performs better on the test set than the training set. However, the test loss fluctuates significantly, particularly during the early and middle stages, which could be due to variations in data distribution or instability during training. Overall, the model's loss decreases steadily, accuracy improves, and the results become stable, demonstrating effective training and convergence.

3.7.3 EfficientB0

Next, we will explore the application of EfficientNet-B0 in this task. As the final model, it shares a similar residual block structure with MobileNetV2. The basic architecture of EfficientNet-B0 has already been introduced, so now let's take a closer look at its practical application.

Like MobileNetV2, we need to use AutoQKeras to search for its values. The figure below shows the results obtained through AutoQKeras. The best configuration has been searched in autoqkeras and presented below:

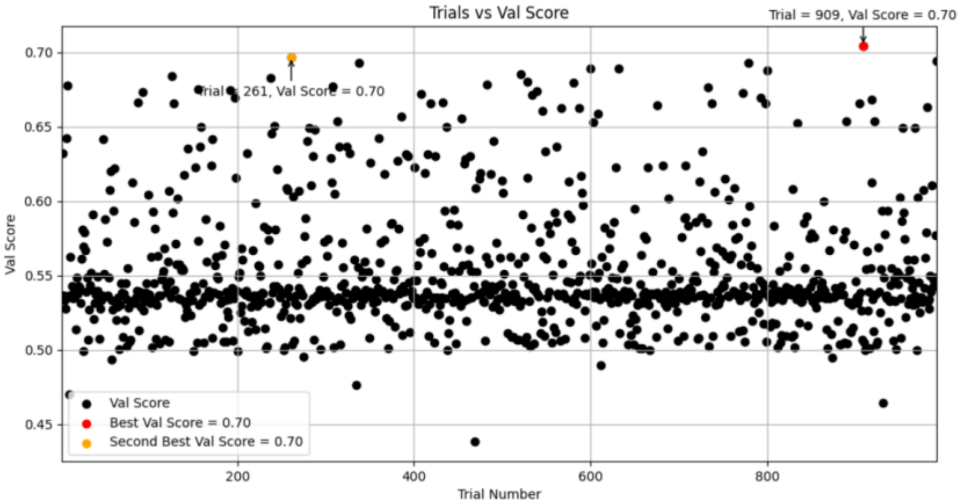


Figure 3.21: searches EfficientNet-B0 scores from autoqkeras over tirals

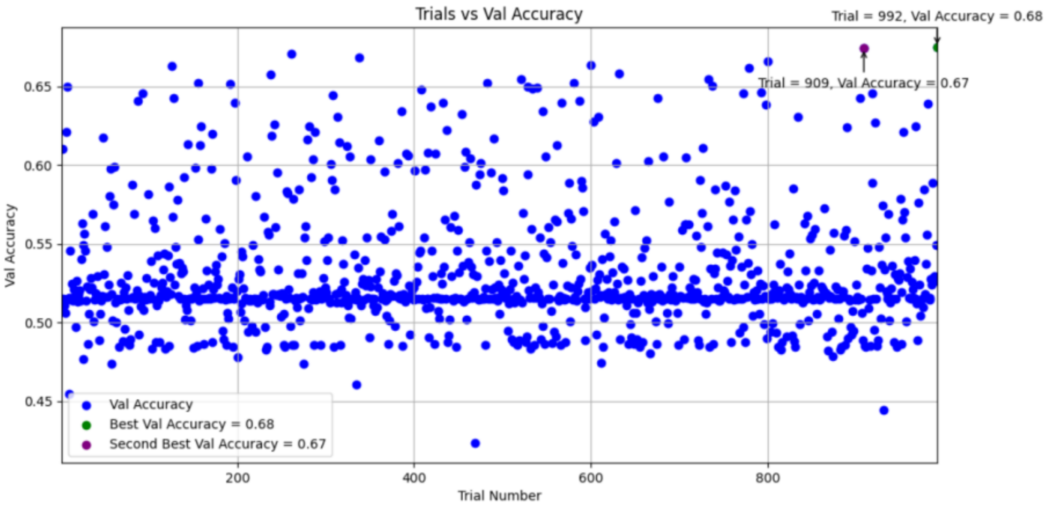


Figure 3.22: searches EfficientNet-B0 accuracy from autoqkeras over tiral

These two charts show that most experiments have validation metrics concentrated in the lower range (around 0.50 to 0.55), with only a few performing better. In the first chart, the best validation accuracy is 0.68 (green, Trial 992), and the second best is 0.67 (purple, Trial 909). In the second chart, the highest validation score is 0.70 (red, Trial 909), with the second highest also being 0.70 (orange, Trial 261). These points stand out from the other experiments. After comprehensive consideration, the results from Trial 909 are the best, so we adopt this trial's results as the optimal configuration.

the internal architecture of trial 909 is:

```
stats: delta_p=0.05 delta_n=0.05 rate=2.0 trial_size=61175468 reference_size=110699218
       delta=4.28%
       a_bits=25625816/46418336 (-44.79%) p_bits=35549652/64280882 (-44.70%)
       total=61175468/110699218 (-44.74%)
activation            quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
stem_conv             f=32 quantized_bits(8,8,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_1          quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
activation_2          quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
block1a_dwconv        f=None quantized_bits(4,4,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_3          quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
activation_4          quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
activation_5          quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
block1a_se_reduce     f=8 quantized_bits(16,16,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_6          quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
activation_7          quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
block1a_se_expand     f=32 quantized_bits(8,8,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_8          quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
activation_9          quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
activation_10         quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
block1a_project_conv  f=16 quantized_bits(16,16,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_11         quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
block2a_expand_conv   f=96 quantized_bits(4,4,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_12         quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
activation_13         quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
block2a_dwconv        f=None quantized_bits(8,8,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_14         quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
activation_15         quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
activation_16         quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
block2a_se_reduce     f=4 quantized_bits(4,4,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_17         quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
```

```
activation_18        quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
block2a_se_expand    f=96 quantized_bits(4,4,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_19        quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
activation_20        quantized_bits_featuremap(4,4,1,1,alpha=’auto’,scale_axis=0)
activation_21        quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
block2a_project_conv f=24 quantized_bits(8,8,1,alpha=’auto’) quantized_bits(16,16,1,alpha=’auto’)
activation_22        quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
block2b_expand_conv  f=144 quantized_bits(4,4,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_23        quantized_bits_featuremap(4,4,1,1,alpha=’auto’,scale_axis=0)
activation_24        quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
block2b_dwconv       f=None quantized_bits(8,8,1,alpha=’auto’) quantized_bits(16,16,1,alpha=’auto’)
activation_25        quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
activation_26        quantized_bits_featuremap(16,16,1,1,alpha=’auto’,scale_axis=0)
activation_27        quantized_bits_featuremap(16,16,1,1,alpha=’auto’,scale_axis=0)
block2b_se_reduce    f=6 quantized_bits(8,8,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_28        quantized_bits_featuremap(16,16,1,1,alpha=’auto’,scale_axis=0)
activation_29        quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
block2b_se_expand    f=144 quantized_bits(4,4,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_30        quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
activation_31        quantized_bits_featuremap(4,4,1,1,alpha=’auto’,scale_axis=0)
activation_32        quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
block2b_project_conv f=24 quantized_bits(4,4,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_33        quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
activation_34        quantized_bits_featuremap(4,4,1,1,alpha=’auto’,scale_axis=0)
block3a_expand_conv  f=144 quantized_bits(8,8,1,alpha=’auto’) quantized_bits(16,16,1,alpha=’auto’)
activation_35        quantized_bits_featuremap(4,4,1,1,alpha=’auto’,scale_axis=0)
activation_36        quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
block3a_dwconv       f=None quantized_bits(8,8,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_37        quantized_bits_featuremap(16,16,1,1,alpha=’auto’,scale_axis=0)
activation_38        quantized_bits_featuremap(4,4,1,1,alpha=’auto’,scale_axis=0)
activation_39        quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
block3a_se_reduce    f=6 quantized_bits(16,16,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_40        quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
activation_41        quantized_bits_featuremap(4,4,1,1,alpha=’auto’,scale_axis=0)
block3a_se_expand    f=144 quantized_bits(16,16,1,alpha=’auto’) quantized_bits(16,16,1,alpha=’auto’)
activation_42        quantized_bits_featuremap(16,16,1,1,alpha=’auto’,scale_axis=0)
activation_43        quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
activation_44        quantized_bits_featuremap(16,16,1,1,alpha=’auto’,scale_axis=0)
block3a_project_conv f=40 quantized_bits(8,8,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_45        quantized_bits_featuremap(16,16,1,1,alpha=’auto’,scale_axis=0)
block3b_expand_conv  f=240 quantized_bits(4,4,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_46        quantized_bits_featuremap(16,16,1,1,alpha=’auto’,scale_axis=0)
activation_47        quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
block3b_dwconv       f=None quantized_bits(4,4,1,alpha=’auto’) quantized_bits(16,16,1,alpha=’auto’)
activation_48        quantized_bits_featuremap(4,4,1,1,alpha=’auto’,scale_axis=0)
activation_49        quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
activation_50        quantized_bits_featuremap(16,16,1,1,alpha=’auto’,scale_axis=0)
block3b_se_reduce    f=10 quantized_bits(8,8,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_51        quantized_bits_featuremap(4,4,1,1,alpha=’auto’,scale_axis=0)
activation_52        quantized_bits_featuremap(16,16,1,1,alpha=’auto’,scale_axis=0)
block3b_se_expand    f=240 quantized_bits(8,8,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_53        quantized_bits_featuremap(16,16,1,1,alpha=’auto’,scale_axis=0)
activation_54        quantized_bits_featuremap(4,4,1,1,alpha=’auto’,scale_axis=0)
activation_55        quantized_bits_featuremap(16,16,1,1,alpha=’auto’,scale_axis=0)
block3b_project_conv f=40 quantized_bits(16,16,1,alpha=’auto’) quantized_bits(16,16,1,alpha=’auto’)
activation_56        quantized_bits_featuremap(4,4,1,1,alpha=’auto’,scale_axis=0)
activation_57        quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
block4a_expand_conv  f=240 quantized_bits(16,16,1,alpha=’auto’) quantized_bits(16,16,1,alpha=’auto’)
activation_58        quantized_bits_featuremap(16,16,1,1,alpha=’auto’,scale_axis=0)
activation_59        quantized_bits_featuremap(4,4,1,1,alpha=’auto’,scale_axis=0)
block4a_dwconv       f=None quantized_bits(8,8,1,alpha=’auto’) quantized_bits(16,16,1,alpha=’auto’)
activation_60        quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
activation_61        quantized_bits_featuremap(16,16,1,1,alpha=’auto’,scale_axis=0)
activation_62        quantized_bits_featuremap(16,16,1,1,alpha=’auto’,scale_axis=0)
block4a_se_reduce    f=10 quantized_bits(16,16,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_63        quantized_bits_featuremap(4,4,1,1,alpha=’auto’,scale_axis=0)
activation_64        quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
block4a_se_expand    f=240 quantized_bits(4,4,1,alpha=’auto’) quantized_bits(16,16,1,alpha=’auto’)
activation_65        quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
activation_66        quantized_bits_featuremap(16,16,1,1,alpha=’auto’,scale_axis=0)
activation_67        quantized_bits_featuremap(16,16,1,1,alpha=’auto’,scale_axis=0)
block4a_project_conv f=80 quantized_bits(4,4,1,alpha=’auto’) quantized_bits(16,16,1,alpha=’auto’)
activation_68        quantized_bits_featuremap(16,16,1,1,alpha=’auto’,scale_axis=0)
block4b_expand_conv  f=480 quantized_bits(8,8,1,alpha=’auto’) quantized_bits(16,16,1,alpha=’auto’)
activation_69        quantized_bits_featuremap(16,16,1,1,alpha=’auto’,scale_axis=0)
activation_70        quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
block4b_dwconv       f=None quantized_bits(16,16,1,alpha=’auto’) quantized_bits(31,31,1,alpha=’auto’)
activation_71        quantized_bits_featuremap(4,4,1,1,alpha=’auto’,scale_axis=0)
activation_72        quantized_bits_featuremap(16,16,1,1,alpha=’auto’,scale_axis=0)
activation_73        quantized_bits_featuremap(8,8,1,1,alpha=’auto’,scale_axis=0)
block4b_se_reduce    f=20 quantized_bits(4,4,1,alpha=’auto’) quantized_bits(16,16,1,alpha=’auto’)
activation_74        quantized_bits_featuremap(16,16,1,1,alpha=’auto’,scale_axis=0)
activation_75        quantized_bits_featuremap(4,4,1,1,alpha=’auto’,scale_axis=0)
```

```
block4b_se_expand    f=480 quantized_bits(4,4,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_76        quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
activation_77        quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
activation_78        quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
block4b_project_conv f=80 quantized_bits(16,16,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_79        quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
activation_80        quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
block4c_expand_conv  f=480 quantized_bits(16,16,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_81        quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
activation_82        quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
block4c_dwconv       f=None quantized_bits(8,8,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_83        quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
activation_84        quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
activation_85        quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
block4c_se_reduce    f=20 quantized_bits(8,8,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_86        quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
activation_87        quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
block4c_se_expand    f=480 quantized_bits(16,16,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_88        quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
activation_89        quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
activation_90        quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
block4c_project_conv f=80 quantized_bits(8,8,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_91        quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
activation_92        quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
block5a_expand_conv  f=480 quantized_bits(16,16,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_93        quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
activation_94        quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
block5a_dwconv       f=None quantized_bits(4,4,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_95        quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
activation_96        quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
activation_97        quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
block5a_se_reduce    f=20 quantized_bits(16,16,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_98        quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
activation_99        quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
block5a_se_expand    f=480 quantized_bits(16,16,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_100       quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
activation_101       quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
activation_102       quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
block5a_project_conv f=112 quantized_bits(16,16,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_103       quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
block5b_expand_conv  f=672 quantized_bits(4,4,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_104       quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
activation_105       quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
block5b_dwconv       f=None quantized_bits(16,16,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_106       quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
activation_107       quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
activation_108       quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
block5b_se_reduce    f=28 quantized_bits(4,4,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_109       quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
activation_110       quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
block5b_se_expand    f=672 quantized_bits(4,4,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_111       quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
activation_112       quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
activation_113       quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
block5b_project_conv f=112 quantized_bits(8,8,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_114       quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
activation_115       quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
block5c_expand_conv  f=672 quantized_bits(4,4,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_116       quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
activation_117       quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
block5c_dwconv       f=None quantized_bits(4,4,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_118       quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
activation_119       quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
activation_120       quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
block5c_se_reduce    f=28 quantized_bits(16,16,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_121       quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
activation_122       quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
block5c_se_expand    f=672 quantized_bits(4,4,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_123       quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
activation_124       quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
activation_125       quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
block5c_project_conv f=112 quantized_bits(8,8,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_126       quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
activation_127       quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
block6a_expand_conv  f=672 quantized_bits(8,8,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_128       quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
activation_129       quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
block6a_dwconv       f=None quantized_bits(16,16,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_130       quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
activation_131       quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
activation_132       quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
block6a_se_reduce    f=28 quantized_bits(8,8,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_133       quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
```

```
activation_134       quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
block6a_se_expand    f=672 quantized_bits(4,4,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_135       quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
activation_136       quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
activation_137       quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
block6a_project_conv f=192 quantized_bits(16,16,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_138       quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
block6b_expand_conv  f=1152 quantized_bits(4,4,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_139       quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
activation_140       quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
block6b_dwconv       f=None quantized_bits(16,16,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_141       quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
activation_142       quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
activation_143       quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
block6b_se_reduce    f=48 quantized_bits(8,8,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_144       quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
activation_145       quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
block6b_se_expand    f=1152 quantized_bits(8,8,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_146       quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
activation_147       quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
activation_148       quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
block6b_project_conv f=192 quantized_bits(16,16,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_149       quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
activation_150       quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
block6c_expand_conv  f=1152 quantized_bits(8,8,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_151       quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
activation_152       quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
block6c_dwconv       f=None quantized_bits(8,8,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_153       quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
activation_154       quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
activation_155       quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
block6c_se_reduce    f=48 quantized_bits(4,4,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_156       quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
activation_157       quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
block6c_se_expand    f=1152 quantized_bits(8,8,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_158       quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
activation_159       quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
activation_160       quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
block6c_project_conv f=192 quantized_bits(4,4,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_161       quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
activation_162       quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
block6d_expand_conv  f=1152 quantized_bits(8,8,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_163       quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
activation_164       quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
block6d_dwconv       f=None quantized_bits(16,16,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_165       quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
activation_166       quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
activation_167       quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
block6d_se_reduce    f=48 quantized_bits(16,16,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_168       quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
activation_169       quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
block6d_se_expand    f=1152 quantized_bits(8,8,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_170       quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
activation_171       quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
activation_172       quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
block6d_project_conv f=192 quantized_bits(8,8,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_173       quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
activation_174       quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
block7a_expand_conv  f=1152 quantized_bits(4,4,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_175       quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
activation_176       quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
block7a_dwconv       f=None quantized_bits(4,4,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_177       quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
activation_178       quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
activation_179       quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
block7a_se_reduce    f=48 quantized_bits(4,4,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_180       quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
activation_181       quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
block7a_se_expand    f=1152 quantized_bits(4,4,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_182       quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
activation_183       quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
activation_184       quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
block7a_project_conv f=320 quantized_bits(4,4,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_185       quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
top_conv             f=1280 quantized_bits(16,16,1,alpha='auto') quantized_bits(31,31,1,alpha='auto')
activation_186       quantized_bits_featuremap(8,8,1,1,alpha='auto',scale_axis=0)
activation_187       quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
activation_188       quantized_bits_featuremap(4,4,1,1,alpha='auto',scale_axis=0)
predictions          u=2 quantized_bits(16,16,1,alpha='auto') quantized_bits(16,16,1,alpha='auto')
activation_189       quantized_bits_featuremap(16,16,1,1,alpha='auto',scale_axis=0)
```

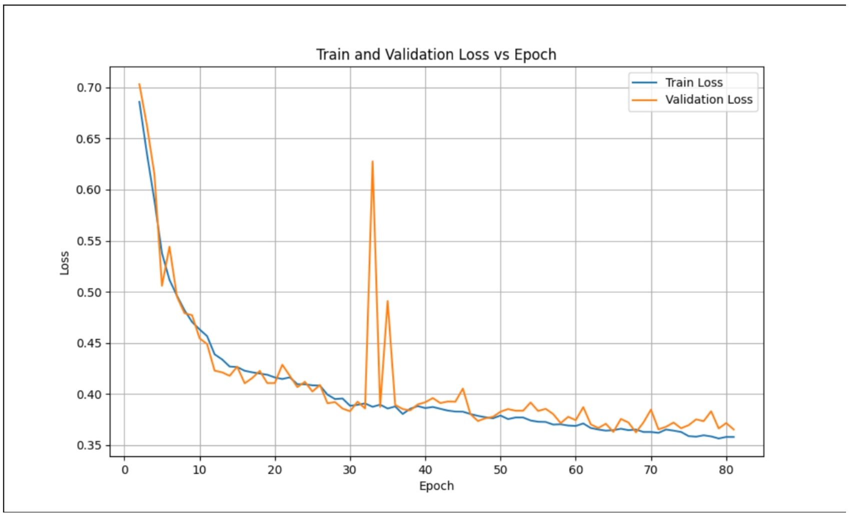


Figure 3.23: EfficientNet-B0 Train and Validation Loss vs Epochs

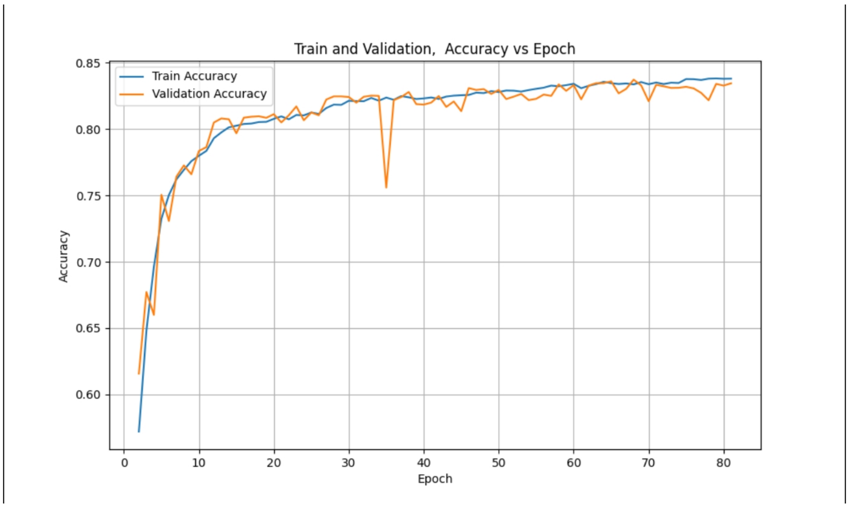


Figure 3.24: EfficientNet-B0 Train and Validation Accuracy vs Epochs

The results align well with expectations. Both training and validation accuracy show a smooth and consistent rise, especially in the later stages where they come together, signaling balanced performance between the two phases. This suggests the model is learning effectively without overfitting or underfitting. While there are a few dips and spikes in the validation accuracy, such fluctuations are typical, often due to the nature of validation batches.

Similarly, the loss curves for both training and validation display a steady decline, indicating the model is continuously improving. The occasional jumps in validation loss suggest momentary instability, likely from data variability or hyperparameter tuning, but the overall trend remains downward, which is promising. In short, the model is progressing well, with stable and healthy performance throughout the training.

Chapter 4

Conclusion and Future Directions

4.1 Summary of Key Findings

This research explores the development of quantized deep learning models for edge applications, focusing on Mixed-Precision Quantization (MPQ) and the use of QKeras in conjunction with AutoQKeras. The key findings from the study include the following:

1. Efficient Quantization with Minimal Accuracy Loss: The application of MPQ allowed for significant reductions in computational complexity and memory requirements without a substantial decrease in model accuracy. By strategically assigning lower precision to less critical layers and using higher precision where necessary, the models maintained a minimal accuracy drop (approximately 1

2. Power of Two Scaling for Efficiency: The implementation of the "auto_po2" scaling mode, which constrains scale factors to powers of two, notably enhanced computational efficiency. This approach transformed costly multiplication and division operations into bitwise shifts, accelerating inference times while preserving model performance. However, challenges arose due to the restricted range of scaling factors, which led to increased sparsity in some models during training.

3. Impact of Quantization-Aware Training (QAT): QAT was successfully integrated into the models to enable them to adapt to low-precision arithmetic, reducing performance degradation typically observed in post-training quantization. The models demonstrated robustness in accuracy, particularly in complex datasets, when subjected to quantized inference.

4. Model Comparisons and Benchmarking: The study evaluated six different models (including MobileNetV1, MobileNetV2, EfficientNet-B0, ResNet, and a CNN) across various quantization settings. MPQ configurations outperformed flat quantization approaches in terms of both accuracy and resource efficiency. EfficientNet-B0 was highlighted as achieving an optimal balance between computational load and predictive performance, particularly in resource-constrained environments.

5. Flat Quantization Performance in Auto_po2: The use of flat quantization under the auto_po2 setting resulted in significantly poor performance. When regularization was applied, it introduced sparsity issues, further degrading the model's effectiveness. In the absence of regularization, although the sparsity problem was avoided, the overall performance remained subpar, highlighting the limitations of flat quantization in this context.

4.2 Current Challenges

While the auto_po2 mode offers computational efficiency by replacing multiplication operations with bitwise shifts, its application in flat quantization leads to significant performance drawbacks, especially when combined with regularization. Even in the absence of regularization, though sparsity is mitigated, models still face challenges in delivering optimal performance.
These issues are not unique to MobileNetV1. Similar performance declines have been observed in other architectures, including ResNet, MobileNetV2, and EfficientNetB0, when flat quantization is paired with auto_po2 mode. The consistent occurrence of these limitations across various models underscores the fundamental constraints of this configuration.
I will use MobileNetV1 as an example to illustrate this.

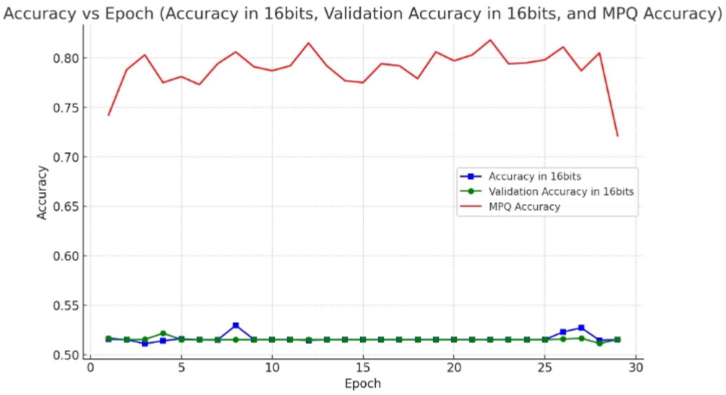


Figure 4.1: train and valication accuracy over epoch in auto_po2 16bits flat quantization, comapared with MPQ

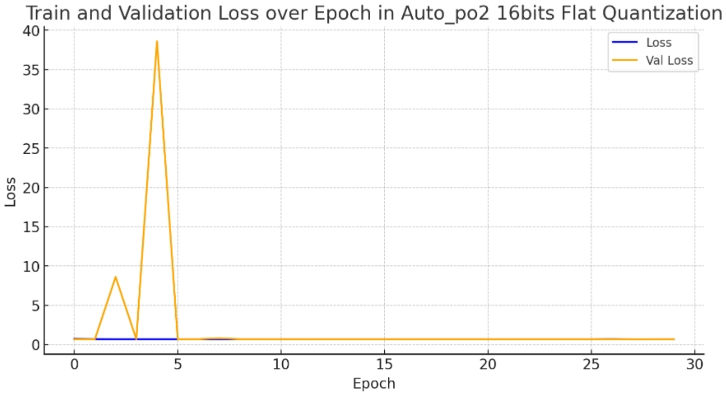


Figure 4.2: train and valication loss over epoch in auto_po2 16bits flat quantization

In these two charts, it can be observed that the training and validation loss for **16-bit flat quantization** showed significant fluctuations during the first few epochs, with the validation loss reaching a high value close to 40 around the 4th epoch. Afterward, the loss quickly dropped and stabilized near zero. At the same time, both training and validation accuracy remained almost flat throughout the process, staying around 50% without noticeable improvement. In contrast, MPQ demonstrated significantly higher accuracy from the start, maintaining a stable performance between 75% and 80%, showing a clear advantage.

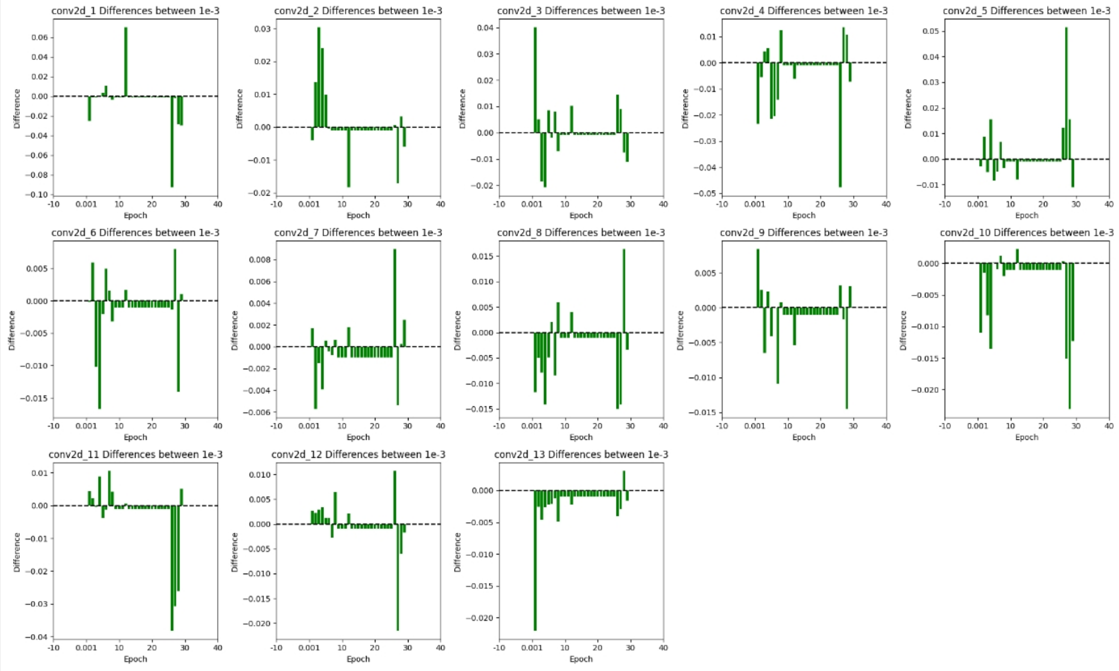


Figure 4.3: updates compared to 1-e3 in every Qconv2d layer over epochs

From the chart, it is evident that some convolutional layers exhibit clear **vanishing gradients** and **exploding gradients** during different epochs of training. For instance, layers like ‘conv2d_6‘, ‘conv2d_7‘, and ‘conv2d_10‘ show almost no weight changes after 20 epochs, indicating vanishing gradients, where the gradients become so small that they fail to update the weights effectively. Meanwhile, layers such as ‘conv2d_1‘, ‘conv2d_5‘, and ‘conv2d_11‘ experience significant weight fluctuations in the early stages of training (especially in the first 10 epochs) and in some mid-epochs, suggesting the presence of exploding gradients, where the updates become excessively large and unstable. As training progresses, weight changes across most layers decrease, with many falling within the range of the yellow lines (±0.0001), further exacerbating the vanishing gradient issue. These phenomena are likely related to the depth of the network, the limitations of quantization precision, and the training optimization strategies used.
In the training stage with ‘auto_po2‘, although the actual calculations are performed in floating-point precision, the quantizer applies a "quantization-dequantization" process with each update, introducing quantization errors. ‘auto_po2‘ restricts the scaling factor to powers of two, meaning that after quantization, weights or activations will carry some error due to the inflexibility of the scaling factor. When weight updates are very small (such as in later training stages), these fine changes might not be effectively quantized, leading to insufficient updates and resulting in vanishing gradients. Conversely, when weight changes are large, dequantization errors may cause updates to become unstable, leading to gradient explosions. Since ‘auto_po2‘ lacks the precision of the ‘auto‘ mode, small updates may be overly magnified or diminished due to the scaling factor limitation, causing gradient issues. The quantizer’s repeated quantization and dequantization during weight updates accumulates these errors over the course of training, resulting in alternating occurrences of gradient explosion and vanishing gradients, as seen in the charts.
However, The key point is why the training converges in MPQ, but does not in flat quantization, despite using the exact same settings.
Unfortunately, since the exact reason hasn’t been discovered, the future work will be addressed in the next chapter.

4.3 Future Prospects in Neural Network Quantization

4.3.1 identify the issue and find the solution

1. **Layer-wise Experiments and Mode Comparison**
   Introduce different quantization modes (such as MPQ and flat quantization) for individual layers to explore the impact of multi-mode combinations on model stability. Specifically, investigate whether using the auto mode for critical layers improves precision, while continuing to use auto_po2 for other layers to maintain computational efficiency:
   Perform layer-wise experiments with a combination of auto and auto_po2 modes in the model and analyze whether this balance improves training stability while maintaining hardware efficiency. Focus on identifying layers where gradient performance improves with different modes, and whether the mixed-mode approach effectively resolves gradient issues.
   Compare the results of using auto_po2 globally versus using a mixed-mode approach, and explore the advantages of multi-mode quantization methods.

2. **Bit-width and Scaling Factor Flexibility Experiments**
   For layers with varying quantization requirements, the research can explore the possibility of dynamically adjusting scaling factors and bit-widths. The experiment should introduce different bit-widths and scaling factors for each layer and assess their impact on gradient stability:

   Investigate methods for dynamically adjusting the scaling factor during training to accommodate the changing needs of small or large gradients. Specifically, introducing an offset or adjustment mechanism could make the scaling factor more flexible.
   For layers with significant quantization errors, attempt to introduce different quantization strategies, adjusting the precision or introducing wider bit-widths, to assess whether mitigating quantization errors can effectively reduce the occurrence of gradient explosions or vanishing gradients.

3. **Quantization Error Evaluation and Comparative Study**
   In the auto_po2 mode, the quantization of weights and activations is constrained by scaling factors restricted to powers of two. This limitation may lead to the accumulation of quantization errors. The research should focus on evaluating quantization errors under different bit-widths and scaling factors:
   Study the quantization and dequantization process, especially the distortion of small gradient updates within the quantizer. It's important to clarify how

quantization errors accumulate during backpropagation and how they affect gradient propagation.newline Conduct comparative experiments with the auto mode to evaluate the precision differences across layers between the two quantization methods. Focus on determining if there is a significant difference in quantization errors and whether these differences are responsible for gradient issues.

4.4 Final Thoughts

This thesis has explored the potential of Mixed-Precision Quantization (MPQ) for deep neural networks, particularly in edge applications. Through extensive benchmarking and analysis, the effectiveness of various quantization strategies, including the innovative use of powers-of-two (po2) scaling factors, was demonstrated in reducing computational complexity without sacrificing significant accuracy. By employing QKeras and AutoQKeras, the models were efficiently optimized for resource-constrained environments, making them well-suited for deployment on embedded systems and mobile devices.

However, the challenges faced during the training process, such as the trade-off between efficiency and model performance, highlight the ongoing need for further research and refinement. The constraints imposed by quantization, particularly in power-of-two scaling, present both opportunities and limitations that must be addressed in future work.

In conclusion, this study has contributed valuable insights into the field of neural network quantization and its application in edge computing. As advancements in hardware continue, MPQ holds great promise for future developments in machine learning models, offering a pathway toward more efficient and scalable AI solutions in the years to come.

[empty]

List of Figures

Bibliography

[1] Kumar D. Biological and Artificial Neural Networks: Computational Models and Python Implementation. https://www.linkedin.com/pulse/biological-artificial-neural-networks-computational-models-dahal-/.

[2] Abdelouafi Boukhris,Antari Jilali,Hiba Asri. "Deep Learning and Machine Learning Based Method for Crop Disease Detection and Identification Using Autoencoder and Neural Network." IIETA.2023

[3] Avadhut Varvatkar. Activations Functions in Deep learning. https://www.kaggle.com/code/avadhutvarvatkar/activations-functions-in-deep-learning

[4] Pranam Shetty. ReLU: A Short Overview on the most Popular Activation Function. https://medium.com/@prxshetty/relu-a-short-overview-on-the-most-popular-activation-function-36d70af84808.

[5] Ben Dickson.What are artificialneural networks (ANN). https://bdtechtalks.com/2019/08/05/what-is-artificial-neural-network-ann/

[6] kamalakkannan r.Neural networks https://www.linkedin.com/pulse/neural-networks-kamalakkannan-r-1ysac/ Conference Name. Publisher

[7] Shubham Kumar.Understanding the Basics Of Artificial Neural Network.

[8] Emmanuel Ohiri.What is a neural network? https://www.cudocompute.com/blog/what-is-a-neural-network

[9] Gradient Descent Algorithm: How Does it Work in Machine Learning? https://www.analyticsvidhya.com/blog/2020/10/how-does-the-gradient-descent-algorithm-work-in-machine-learning/.

[10] Vipin. Neural Networks: Everything You Should Understand. https://medium.com/@vipinra79/neural-networks-everything-you-should-understand-6b99c20ccd5c

[11] Nafiz Shahriar.What is Convolutional Neural Network — CNN (Deep Learning) https://nafizshahriar.medium.com/what-is-convolutional-neural-network-cnn-deep-learning-b3921bdd82d5.

[12] Sabina Pokhrel. Beginners Guide to Convolutional Neural Networks

https://towardsdatascience.com/beginners-guide-to-understanding-convolutional-neural-networks-ae9ed58bb17d.

[13] Kunlun Bai.A Comprehensive Introduction to Different Types of Convolutions in Deep Learning .https://towardsdatascience.com/a-comprehensive-introduction-to-different-types-of-convolutions-in-deep-learning-669281e58215.

[14] Chi-Feng Wang. A Basic Introduction to Separable Convolutions. https://towardsdatascience.com/a-basic-introduction-to-separable-convolutions-b99ec3102728.

[15] ResNet iamge(japanese version) zero2one,https://zero2one.jp/ai-word/resnet/

[16] Paul-Louis Pröve,MobileNetV2: Inverted Residuals and Linear Bottlenecks ,https://towardsdatascience.com/mobilenetv2-inverted-residuals-and-linear-bottlenecks-8a4362f4ffd5

[17] Mark Sandler,Andrew Howard,etc. MobileNetV2: Inverted Residuals and Linear Bottlenecks. google, 2019.

[18] Kenichi Nakanishi.Exploring Convolutional Neural Network Architectures with fast.ai. https://towardsdatascience.com/exploring-convolutional-neural-network-architectures-with-fast-ai-de4757eeeebf .

[19] Vishal Rajpu. Efficient Nets: Scaling of Conv networks. https://medium.com/aiguys/efficient-nets-scaling-of-conv-networks-6ffa0e102fc7.

[20] Mingxing Tan,Quoc V. L.fficientNet-B0 baseline network. Publisher, 2020.

[21] An Introduction to Quantization in Deep Learning: Theory Edition(japanese version). https://tech.retrieva.jp/entry/20220128.

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

[23] Luca Urbinati.Qkeras-mod-explained https://github.com/LucaUrbinati44/qkeras-mod/blob/main/qkeras-mod-explained.ipynb, 2024.

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

[25] Marco Terlizzi. Mixed-Precision Quantization and Inference of MLPerf Tiny DNNs on Precision-Scalable HardwareAccelerator. polito, 2023.

[26] Fatemeh Esmaeili , et al. Anomaly Detection for Sensor Signals Utilizing Deep LearningAutoencoder-Based Neural Networks. 2023.

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

[28] Emmanuel Deruty. Intuitive understanding of MFCCs. Medium.2022

[29] Authors, et al. "Title." Conference Name. Publisher, YYYY.

[30] Silicon Labs. Person detection.
https://github.com/SiliconLabs/

[31] Luca Urbinati and Marco Terlizzi. Visual wake words test set github issue. 79 Bibliography

[32] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. Microsoft coco: Common objects in context, 2015

[33] Vijay Janapa Reddi, et al. MLPerf Inference Benchmark,2020

[34] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The CIFAR-10 dataset