Reading Summary Week 5

Aakash Agrawal
HDSI
University of California San Diego
San Diego, CA, 92092
aaa015@ucsd.edu

1 A Survey of Quantization Methods for Efficient Neural Network Inference

The paper begins with a simple introduction to the problem of quantization, which essentially asks how continuous real-valued numbers can be mapped to a discrete set of values in a way that minimizes the bit requirements, without significantly compromising accuracy.

The authors argue that most neural network models are **over-parameterized**. While the performance of these deep learning models has gradually improved, this has come at the cost of their sheer size, making them impractical for deployment in resource-constrained environments where real-time inference, low energy consumption, and high accuracy are key. The authors outline several areas of work besides Quantization that aim at making neural networks more efficient, covering aspects of design, training, and deployment.

- **Designing Efficient NN Architectures**: This area of research focuses on improving the efficiency of neural networks by developing novel architectures. However, significant advancements in this direction typically rely on manual search, which is not scalable. As a result, there has been a growing focus on Automated Machine Learning (AutoML) and Neural Architecture Search (NAS), which aim to automatically discover the optimal architecture while considering constraints such as model size, depth, and width.
- **Co-Designing NN Architectures and Hardware**: Since the overheads of neural networks, such as latency and energy consumption, are heavily dependent on hardware, this area of research focuses on adapting neural network architectures for specific target hardware platforms, ensuring optimal performance and efficiency.
- **Pruning**: In pruning, neurons that have a small impact on the output or loss function are removed, leading to a much sparser neural network model. Pruning methods are categorized into:
  - **Unstructured Pruning**: In this method, neurons that don't significantly contribute to the output are removed wherever they occur in the network. Aggressive unstructured pruning typically doesn't hurt the model's generalization ability. However, it can result in sparse matrix operations, which are hard to accelerate and are often memory-bound.
  - **Structured Pruning**: Here, entire groups of parameters, like a convolution block, are removed. This approach still allows for efficient dense matrix multiplications. However, aggressive structured pruning can lead to a significant drop in accuracy.
- **Knowledge Distillation**: This method involves training a large model first and then using it as a "teacher" to train a smaller, more compact "student" model. Knowledge distillation methods can result in noticeable accuracy loss when the compression is too aggressive.

1.1 Basic Concepts of Quantization

Goal of Quantization: Reduce the precision of both model parameters (θ) as well as the intermediate activation maps to a low-precision alternative that has a minimal impact on the performance of the model.

1.1.1 Uniform Quantization

In uniform quantization, the distances between the quantized values (or levels) are the same. The formula for uniform quantization is given by: $Q(r) = Int(r/S) - Z$, where Q is the quantization operator, r is a real-valued input, S is a scaling factor, and Z is the zero-point.

The choice of the scaling factor (S) depends on a clipping range given by $[\alpha, \beta]$ and a desired quantization bit width b.

$$S = \frac{\beta - \alpha}{2^b - 1}$$

The process of choosing the clipping range is known as **calibration**. The choice of α, β leads to two different types of quantization:

- **Symmetric Quantization**: Here, the clipping range is symmetric around the origin, i.e., $\alpha = -\beta$, implying that the zero-point is zero. This quantization is widely adopted as it reduces computational cost during inference by setting the zero-point to zero, which simplifies the implementation and is also straightforward to implement.
- **Asymmetric Quantization**: Here, the clipping range is not necessarily symmetric around the origin, i.e., the zero-point is non-zero. Asymmetric quantization is desired when the weights or activations are imbalanced or skewed, e.g., the activation after ReLU always has non-negative values. Further, it is to be noted that the zero point here is static and data/input independent; hence, it can be absorbed into the bias term during inference.

How to choose α, β?

- Using **min/max** values: We can set $\alpha = r_{min}$ and $\beta = r_{max}$. However, this approach is sensitive to outliers in the activations, which could unnecessarily increase the range and reduce the quantization resolution.
- Using **percentile** values: We can use the i-th largest/smallest values as β/α.
- Minimizing **KL-divergence**: We can minimize the information loss between real values and the quantized values.

1.1.2 Range Calibration Algorithms

An important question to ponder is: "**When** the clipping range is determined?". Since the weight parameters are fixed during the inference, the clipping range can be statically computed for weights. However, the activation maps vary from input to input, which leads to two approaches for quantizing activations:

- **Dynamic quantization**: The clipping range is dynamically computed at runtime for each activation. This approach incurs a very high compute overhead as it requires real-time computation of stats like min, max, percentile, etc. However, it leads to better accuracy.
- **Static quantization**: Here, the clipping range is pre-computed and is static during inference. Although, it doesn't add any compute overhead, it results in lower accuracy as compared to the dynamic counterpart.

1.1.3 Quantization Granularity

Quantization methods vary based on how the clipping range for the weights is calculated.

- **Layerwise Quantization**: In this method, a single clipping range is used for all the convolutional filters (or kernels) in a specific layer. It's simple to implement, but it can result in suboptimal accuracy because the range of each filter in the layer can vary greatly.
- **Groupwise Quantization**: This approach groups multiple channels within a layer to calculate the clipping range. It's helpful when the parameter distribution in a single channel varies a lot, leading to inefficient use of quantization space.
- **Channelwise Quantization**: Here, each convolutional filter in a layer has its own independent clipping range. This method offers better quantization resolution and generally results in higher accuracy.

- **Sub-channelwise Quantization**: This method goes even further by determining the clipping range for groups of parameters within a single channel. While it can improve accuracy, it introduces significant compute overhead because different scaling factors must be considered for each filter in the layer.

1.1.4 Non-uniform Quantization

In non-uniform quantization, the quantization levels and steps are allowed to be non-uniformly spaced. The formula for non-uniform quantization is given by: $Q(r) = X_i$, if $r \in [\Delta_i, \Delta_{i+1})$, where X_i represents the quantization levels, Δ_i represents the quantization steps. Non-uniform quantization can achieve higher accuracy for a given bit-width, as it allows for better capturing of the distributions by concentrating on important value regions or selecting suitable dynamic ranges. However, they are difficult to deploy in production.

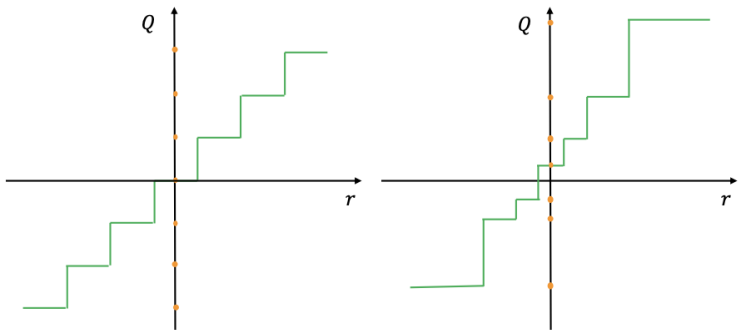

Figure 1: Linear v/s Non-linear quantization.

1.1.5 Quantization-Aware Training (QAT)

After quantization, it's often necessary to adjust the parameters in the neural network to maintain or improve performance. QAT is a powerful technique that helps neural networks retain high accuracy in low-precision formats. Here, the forward and backward passes are performed on the quantized model in floating point, but the model parameters are quantized after each gradient update. Performing the backward pass in floating point is crucial because accumulating gradients in quantized precision can lead to zero gradients or large errors, especially in low-precision settings.

A key challenge in QAT is handling backpropagation, as the quantization and rounding operations are non-differentiable. To address this, a technique called the **Straight Through Estimator** (STE) is used, which applies a coarse gradient approximation. STE essentially bypasses the rounding operation during the backward pass by treating it as an identity function, allowing gradients to flow through as if no quantization occurred. This enables effective weight updates while still accounting for quantization effects.

Another approach to bypass the non-differentiability of the quantization operator is to use regularization to gradually enforce weight quantization. These methods, known as **Non-STE** methods, avoid the STE approximation but often require extensive tuning for optimal performance.

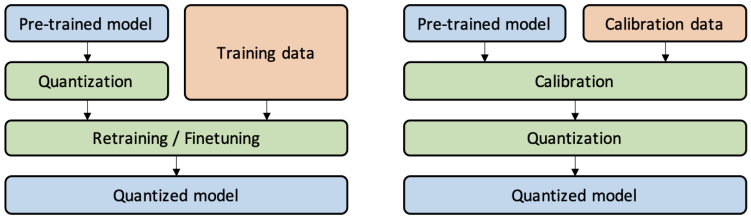

Figure 2: Quantization-Aware Training (QAT) v/s Post-Training Quantization (PTQ)

1.1.6 Post-Training Quantization (PTQ)

In PTQ, a pre-trained model is fine-tuned using a small set of calibration data to find the best clipping ranges and scaling factors. Once calibrated, the model is quantized to lower precision while keeping accuracy reasonably intact. PTQ is quick to implement and works well even when data is limited or unlabeled.

1.1.7 Zero-shot Quantization (ZSQ)

Zero-shot (aka data-free) quantization is an extreme form of post-training quantization (PTQ) where no training or testing data is used during the quantization process. This approach is useful in situations where access to training data is not possible, such as due to privacy concerns, legal restrictions, or when the dataset is too large to distribute.

In this approach, synthetic data is used to fine-tune the quantized model. The synthetic data must closely resemble the real data for the approach to be effective. One way to generate this synthetic data is by using **Generative Adversarial Networks** (GANs). In this method, a pre-trained model is used as a discriminator to train a generator. The generator then produces synthetic data samples, which can be used to fine-tune the quantized model. Knowledge distillation is often used to transfer knowledge from the full-precision model to the quantized one. This method helps in situations where using real data is not possible, but achieving good performance still matters.

1.1.8 Stochastic Quantization

Generally, the quantization scheme during inference is **deterministic**, i.e., each value is always mapped to the same quantized value without randomness. However, during training, a **stochastic** quantization scheme is preferred, which introduces randomness when mapping and helps the model adapt to quantization noise. However, a major challenge with stochastic quantization is the computational overhead of generating random numbers for every weight update, making it less practical for widespread adoption.

1.2 Quantization below 8-bits

Here, the goal is to drastically reduce the model size, memory requirements, and computational complexity, enabling faster inference and lower power consumption, especially on resource-constrained devices like mobile or embedded systems.

1.2.1 Simulated and Integer-only Quantization

In simulated quantization, model parameters are stored in low precision, but operations like matrix multiplication and convolutions still use floating-point arithmetic. This means the benefits of low-precision logic are not fully utilized because the model parameters need to be dequantized (converted back to higher precision) during computations.

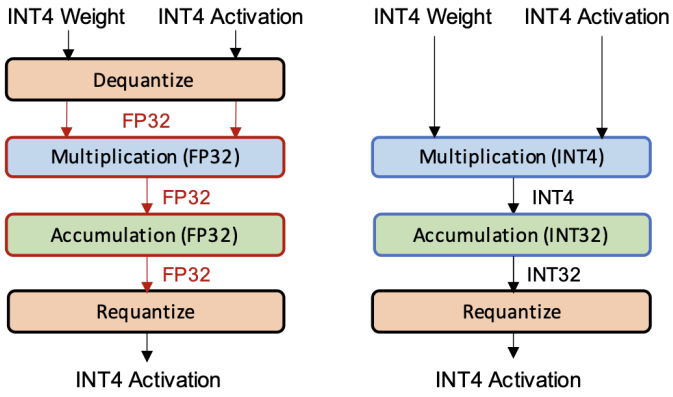

Figure 3: Simulated Quantization v/s Integer-only Quantization

On the other hand, in integer-only quantization, everything—model parameters, activations, and operations—uses low-precision integer arithmetic. This allows the entire inference process to be carried out efficiently using integers without the need for dequantization. This approach has clear advantages in terms of lower latency, reduced power consumption, and better area efficiency compared to using full-precision floating-point operations. This enables the deployment of NNs on low-end microprocessors, which often lack floating-point units.

1.2.2 Mixed-Precision Quantization

Different layers in a neural network have varying impacts on the loss function. Uniformly quantizing the entire model to ultra-low precision can lead to a significant loss in accuracy due to reduced representation and loss of important details. In mixed-precision quantization, each layer is quantized with a different bit precision based on how sensitive it is to quantization. Layers that are more sensitive use higher precision, while less sensitive layers use lower precision. This approach has proven to be effective and efficient for hardware, making it a good choice for low-precision quantization.

1.2.3 Extreme Quantization

Extreme Quantization refers to reducing values to 1-bit (**binarization**) or 2-bit (**ternarization**) representations. This can greatly reduce model size, inference latency, and allow for more efficient computation using bit-wise operations. However, simple binarization or ternarization often leads to significant accuracy loss. To address this, solutions like designing binarization-aware loss functions, minimizing quantization errors with a combination of binary weight matrices, and using improved training methods have been proposed to reduce accuracy degradation.

2 Deep Compression

Building on the issue of over-parameterization in neural networks, researchers have proposed Deep Compression, a three-stage pipeline that combines **pruning**, trained **quantization**, and **Huffman coding**. Deep Compression reduces the storage requirements of neural networks by **35x** to **49x** with minimal impact on performance. This enables models to fit into on-chip SRAM cache instead of relying on off-chip DRAM, significantly improving efficiency. Additionally, this compression technique makes it feasible to deploy complex neural networks in mobile applications, where storage and bandwidth constraints are critical. This has many advantages, including better privacy, reduced network bandwidth usage, and real-time processing.

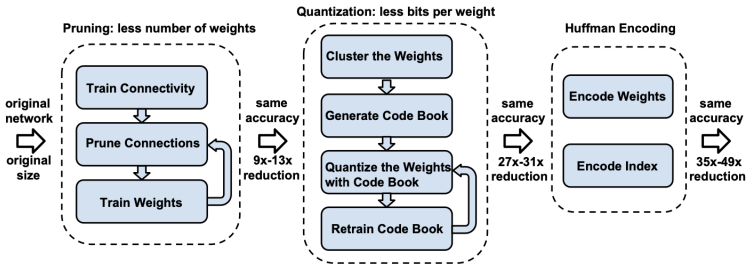

Figure 4: The three stage compression pipeline: pruning, quantization and Huffman coding.

2.1 Pruning

The goal of pruning is to remove redundant connections or those that have little impact on the output or loss function in a trained model, keeping only the most informative connections. After pruning, the network is retrained to adjust the remaining sparse connections, which can then be stored in **CSR** (Compressed Sparse Row) or **CSC** (Compressed Sparse Column) format. This technique reduces the number of parameters by 9× in AlexNet and 13× in VGG-16.

To further improve compression, relative indexing is used instead of absolute indexing. This allows index storage to be more compact, requiring only 8 bits for convolutional layers and 5 bits for fully connected layers.

2.2 Trained Quantization & Weight Sharing

This approach further compresses the pruned network by reducing the number of bits needed to store each weight. It works by limiting the number of unique weights, allowing multiple connections to share the same weight. These shared weights are then fine-tuned during training. The compression rate is given by the formula:

$$r = \frac{nb}{nlog_2(k) + kb}$$

Where k is the number of clusters (or shared weights), n is the number of connections, and each connection is represented with b bits, $log_2(k)$ is the number of bits required to encode the index of each shared weight. This method ensures that each connection only uses k-shared weights, further reducing the storage requirements.

How is weight sharing achieved?
Using **k-means** clustering. Note that all the weights belonging to the same layer share the same weight. Weights are not shared across layers.

Important differences to HashNet?
In HashNet, weight sharing is determined by a hash function before the network sees any training data. In contrast, k-means quantization determines weight sharing after the network is fully trained, allowing the shared weights to approximate the original network more accurately.

2.2.1 Initialization of shared weights

Initialization plays a crucial role in the final convergence of clustering and thereby determines the network's prediction accuracy. There are three types of initialization:

- **Forgy** (random) initialization randomly selects k observations from the dataset as centroids. It tends to concentrate around the two peaks in a bimodal distribution but may not represent large weights well.
- **Density-based** initialization spaces the centroids based on the CDF of weights. This method creates denser centroids around the peaks but is more scattered than the Forgy method.
- **Linear** initialization spaces centroids evenly between the minimum and maximum of the weights, making it invariant to the weight distribution. This method is the most scattered and does not struggle with representing large weights, unlike the other two methods.

Larger weights are important but less frequent, and linear initialization handles them better compared to Forgy and density-based methods.

2.3 Huffman Coding

Huffman coding works by assigning shorter codes to more frequent symbols and longer codes to less frequent ones, reducing the total number of bits needed to store the data. It takes advantage of the **biased** distribution of effective weights in neural networks, where most quantized weights are concentrated around two peaks. By encoding these non-uniformly distributed values efficiently, Huffman coding helps save 20%–30% of network storage. Huffman coding doesn't require training and is implemented offline after all the fine-tuning is finished.

2.4 Experimental Results

The authors present the result of deep compression on four networks: **LeNet-300-100** and **LeNet-5** network on the MNIST dataset and **AlexNet and VGG-16** on ImageNet dataset. Overall, the compression pipeline saves network storage by 35x to 49x across these networks.

- **LeNet-300-100 and LeNet-5**: LeNet-300-100 is a fully connected network with two hidden layers containing 300 and 100 neurons, respectively. After compression, it achieves a **1.6% Top-1 error** on the MNIST dataset, with a **40x** compression. This error rate is very close to that of the original, uncompressed model.

- **AlexNet**: AlexNet can be compressed to **2.88%** of its original size while maintaining the top-1 accuracy of **57.2%** on the uncompressed reference model on the ImageNet dataset, thereby achieving a **35x** compression. The total size of AlexNet decreased from 240MB to 6.9MB. Here, weights in the CONV layers are encoded with 8 bits, FC layers use 5 bits, and the relative sparse indexes are encoded with 4 bits.
- **VGG-16**: The VGG16 network was compressed by **49x** of its original size. The total size of VGG16 decreased from 552MB to 11.3MB. Weights in the CONV layers are represented with 8 bits, and FC layers use 5 bits, which does not impact the accuracy.

2.4.1 Pruning and Quantization

When applied individually, the accuracy of a pruned network drops significantly when compressed below 8% of its original size, and the accuracy of a quantized network also declines when compressed below 8%. However, when both techniques are combined, the network can be compressed to just **3%** of its original size without any loss of accuracy. On the other hand, **SVD**, although inexpensive, has a poor compression rate.

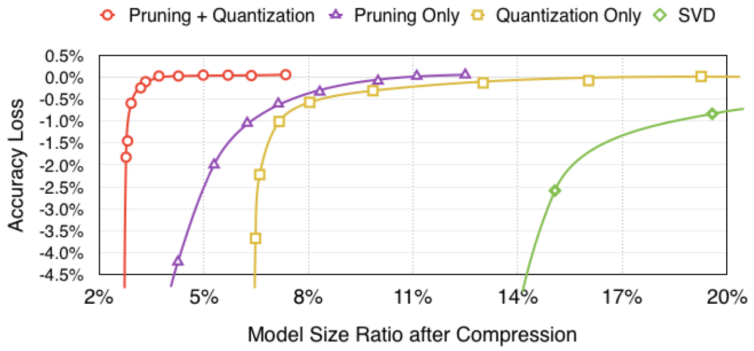

Figure 5: Accuracy v/s compression rate under different compression methods.

Pruning and quantization are most effective when **used together**. For example, unpruned AlexNet has 60 million weights to quantize, whereas pruned AlexNet has only 6.7 million weights. With the same number of centroids, the pruned model introduces less quantization error, leading to better efficiency and accuracy.