Data Science Institute
COLUMBIA UNIVERSITY

Fall 2020 Capstone Project

Progress Report II

Energy-Efficient AI on EDGE

Kumari Nishu (kn2492)

Neelam Patodia (np2723)

Mohit Chander Gulla (mcg2208)

Pritam Biswas (pb2796)

Prasham Dhaneshbhai Sheth (pds2136)

Table of Contents

1. Progress Overview

In the first phase of the project, we focused on trying out various post-quantization techniques to get an understanding of accuracy change with change in model complexity, datasets and techniques (i.e. mid-rise quantization, regular rounding, stochastic rounding) at varying levels of precision. In the second phase of our project, we shifted the focus to explore a breadth of different approaches to model quantization, namely model pruning, quantization-aware training (QAT), and multipoint with mixed precision. Moreover, we modularized our code in different modules based on functionality and we will be adding a basic tutorial notebook to give a walkthrough of how to use the various implementations. Lastly, we aim to open source our work (which can be accessed at **https://github.com/mohitgulla/Edge**) as a python package so that it can be easily used by anyone.

2. Literature Survey

2.1. Post-Training Quantization with Multiple Points: Mixed Precision without Mixed Precision

The research paper, Post-Training Quantization with Multiple Points: Mixed Precision without Mixed Precision *[1]*, as the name suggests examines the post-training quantization problem where we discretize the weights of a pre-trained deep learning model without re-training the model. The authors, Xingchao Liu et al., propose a multipoint quantization method that approximates a full-precision weight vector using a linear combination of multiple vectors of low-bit numbers. This is in contrast to typical quantization methods that approximate each weight using a single low precision number. The idea of "mixed precision" is achieved by adaptively choosing the number of low precision points on each quantized weight vector based on the error of its output but without a physical mixed-precision implementation. For detailed methodology of this paper, please refer to Appendix section 8.1.

The multipoint quantization shows promising results by outperforming various state-of-the-art quantization methods on ImageNet classification. At 8-bit precision, the reduction in model size is significant and the accuracy drop varies with the complexity of the model. Previously, we performed post-training quantization using simpler techniques such as mid-rise quantization, stochastic rounding, etc. We will implement the multi-point quantization algorithm as detailed in this paper, to compare with the post-training quantization techniques previously implemented. We will be extending this idea further by evaluating the effectiveness of multipoint at lower precision bits.

2.2. Unified Framework of DNN Weight Pruning and Weight Clustering / Quantization Using ADMM

Shaokai Ye et al. in their work *[2]* combine the two compression techniques: Pruning, Quantization and develop an unified algorithm to find the optimal setting of weights utilizing the power of both techniques to maximize the exploitation of redundancy in the Deep Neural Networks. They form the problem as a constrained optimization problem and use alternating direction method of multipliers (ADMM)*[3]* to find a set of optimal weights that satisfy the constraints of both

pruning and quantization techniques. Further, they also experiment with clustering as a way to achieve quantization where cluster centroids obtained using clustering would represent the value for all the weights in that particular cluster. For detailed methodology of this paper, please refer to Appendix section 8.2.

The proposed framework initiates a great idea to combine pruning with quantization. Overall, the ADMM-based solution can be understood as a smart, dynamic regularization process in which the regularization target is dynamically updated in each iteration. They have presented the results on AlexNet and LeNet-5; both of which show the benefits of this promising technique. They compare the results across the other works in terms of the memory size required by the network, average number of bits per layer, as well as accuracy degradation. Having read about the technique to find the multi-point precision from the work of Xingchao Liu et al., we decided to combine the pruning techniques with the previously designed post-quantization techniques as well as the multi-point precision quantization technique. We thus formulated the combination of tasks as 2 independent ones as opposed to the implementation in the paper where they combine both into one formulation and solve the problem of finding optimal weights by formulating it into a non-convex optimization problem and solving it using ADMM. This way of formulating into two independent tasks allows us to experiment with pruning separately and then merge with the quantization techniques that are designed separately.

2.3. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference

This research paper from Google *[4]* proposes a quantization scheme to quantize both the weights and activations of a CNN from full precision into INT8. It aims to improve the latency-vs-accuracy tradeoffs of earlier methods on real hardware. It undertakes a more meaningful challenge to quantize the architecture of already efficient model-MobileNets. The reason being it is easier to quantize architectures such as AlexNet and VGG as they are over-parameterized by design in order to achieve marginal gain in accuracy. Their quantized inference framework can be efficiently implemented on an integer- arithmetic-only hardware such as ARM and x86 CPU. For detailed methodology of this paper, please refer to Appendix section 8.3.

They conducted two sets of experiments. One to demonstrate the effectiveness of simulated quantization and other to illustrate the improved accuracy-vs-latency tradeoff. Our interests for the capstone project are more aligned with the former approach. They have presented the results on ResNet50 and Inception-v3; both of which show the benefits of this promising technique. They have compared their results with those of binary weight networks , ternary weight networks, incremental network quantization and fine-grained quantization. We would be implementing this technique to compare with the post-training quantization techniques previously implemented. We will be extending this idea further by evaluating its effectiveness at lower precision bits and adopting different quantization schemes.

2.4. HAQ: Hardware-Aware Automated Quantization with Mixed Precision

Model quantization is a widely used technique to compress a trained deep neural network (DNN) model in order to decrease the inference time. With emergent complex and bigger neural network architecture, DNN hardware accelerators begin to support mixed precision (1-8 bits) - Apple released the A12 Bionic chip that supports mixed precision, NVIDIA introduced the Turing GPU architecture that supports 1-bit, 4-bit, 8-bit and 16-bit arithmetic

operations. But Leveraging such specialized architectures poses a big challenge - how to find the optimal bit width for each layer to quantize. Conventional quantization algorithms ignore the different hardware architectures and quantize all the layers in a uniform way. The paper *[5]* introduces the Hardware-Aware Automated Quantization framework which leverages reinforcement learning to automatically determine the quantization policy (optimal bit width for different layers) and take the hardware accelerator's feedback in the design loop. For detailed methodology of this paper, please refer to Appendix section 8.4.

Training an RL agent for a DNN architecture and a hardware configuration

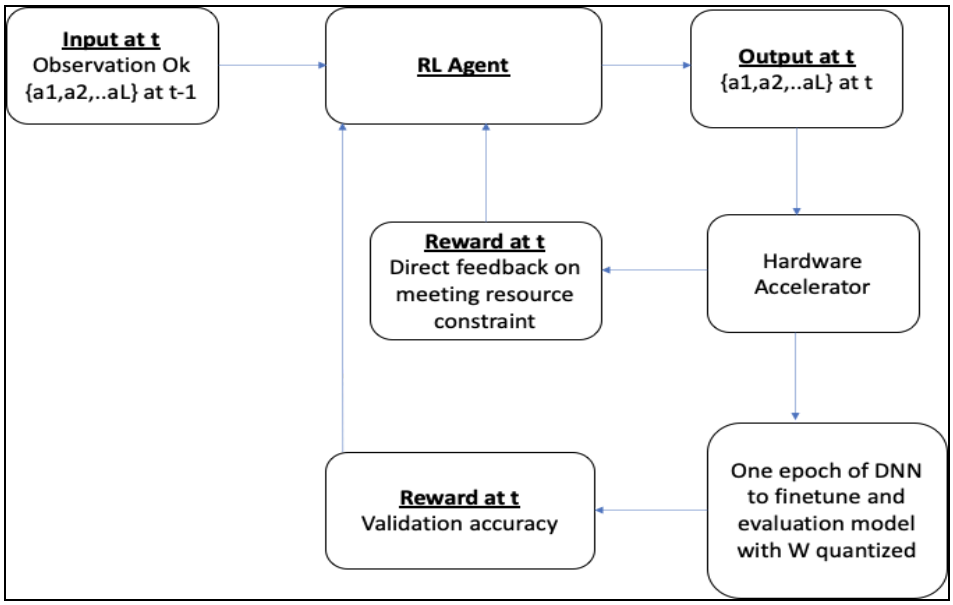


Fig 1: Optimal Quantization Policy Using RL agent

Overall, this paper provides an innovative approach to learn the optimal bit width to quantize different layers differently and gain the maximum inference accuracy. But the RL training process leveraged the hardwares to estimate the validation accuracy of the quantized model and the direct feedback on the resource used by the quantized model in terms of inference latency, energy used and the model size. Hence, involvement of the hardware in the training process was a key component of the RL training process. Due to this hardware limitation, we decided to move ahead with another post training quantization approach for our project.

3. Implementation Details

3.1. Multipoint Post Training Quantization

Inspired by the paper, *Post-Training Quantization with Multiple Points: Mixed Precision without Mixed Precision*, we have implemented multipoint quantization as a post-training quantization technique. Moreover, modifications to the algorithm have been made to improve processing time as well as quantization performance. We want to evaluate whether a novel idea of multipoint precision quantization, with the flexibility it offers, outperforms some of the simpler

post-training quantization methods like stochastic rounding and mid-rise quantization. The pseudo code of our implementation is detailed below:

PSEUDOCODE FOR MULTIPOINT OPTIMIZATION

---

1. func optimize(weights, n, eta, qset):
2. Initialize $r_1 = weights$
3. for $i = 1 \ : \ n$ do:
4. Compute Δ_r , the minimal gap of r_i
5. Set step size γ and search space I as given by equation (i)
6. Using grid search solve equation (ii) to find a_i^*
7. Set $w_i^* = [r_i / a_i^*]_Q$, $r_{i+1} = r_i - a_i^* w_i^*$
8. return $\{a_i^*, \ w_i^*\}_{i=1 \ldots n}$

The minimal gap, Δ_{r_i} is the minimal distance between two elements in the weight vector. It restricts the maximum value of the step size. Grid search enumerates all the values from set $[I_{min} : \gamma : I_{max}]$, to select the value that archives the lowest error. The choice of search range, I (eq. i) and step size, γ (eq. ii) are critical. The paper proposes the following setting:

$$I = [0, \ 2(2^{b-1} - 1) \ ||r_i|| \ ] \qquad \text{(i)}$$

$$\gamma = min(\Delta_{r_i} / (2^{b-1} - 1), \ \eta) \qquad \text{(ii)}$$

PSEUDOCODE FOR MULTIPOINT QUANTIZATION

---

1. func multipoint_quantization(model, calibration_data):
2. Run a forward pass of model on calibration_data to get layer wise input activations
3. for layer in model.layers(except for the first and last one):
4. K = $max(weights)$, B = $mean(weights)$ // naive implementation, without clipping
5. $qset$ = K + $[-1, 1]$ uniform grid with $1/(2^{p-1} - 1)$ step size + B // set of possible quantized values
6. $weights' = round(weights, \ qset)$ // round full-precision weight to nearest quantized weight
7. $\{a^*, \ w^*\}_{i=1 \ldots n} = optimize(weights', \ n, \ \eta, \ qset)$ // linear combination of weights
8. $weights = \sum_{i=1}^{n} a_i^* w_i^*$ // reassign quantized weights to the model
9. return quantized_model

As we quantize our fully-trained model, we are skipping the first and the last layer. The underlying assumption here is that these two layers contain a lot of important information and quantizing them would drastically reduce the accuracy of our model. The crux of the algorithm is to find optimal values of a^* and w^*, which allows us to form linear combinations to represent weights of each layer. We have implemented a naive version of the algorithm presented in the paper where we fix n and use the same set $\{a^*, \ w^*\}_{i=1 \ldots n}$ to represent all layers. To further improve quantization performance and make our algorithm more flexible, we need to learn the value of n. This is done by computing the approximation error between full precision and quantized weights on calibration data. We would start with $n = 1$ and keep increasing n till our error for a layer falls below a threshold, ε. As setting an appropriate value of ε requires running a lot of experiments

as it changes with model complexity as well as data, we proceeded with a naive implementation and set $n = 10$ in our implementation.

3.2. Post Training Pruning

Pruning- a technique to zero out certain nodes or connections in the network which allows us to compress the model as well as gives us quicker and less complex computations while inference time. The method is majorly targeted towards removing the less contributing (i.e.) unnecessary connections from the network. As these connections would be zeroed out and hence while storing we can skip storing these connections completely, while inference time we can completely omit out the calculations from that stage resulting in reduced computations.

Pruning can be achieved using various heuristics allowing us to decide the importance of a particular node/connection for the model's performance. There are various ways to prune the network (like in the work of Song et al. *[6])* based on different heuristics and some of the methods are available to use from PyTorch's implementation *[6]* as well as Keras's implementation *[7]*. While using the existing library functions allows us to achieve the goal for pruning it doesn't allow us the flexibility to play around with the heuristics. Thus, we decided to implement the pruning part from scratch using the heuristic: the lesser magnitude of the weight; the lesser is its importance towards the model's performance.

Assumptions made while implementing the pruning algorithm:

- We assume that the weights of the last layer should not be pruned because the last layer is generally the task specific layer and predicts the probability of each class. If we prune the weights from the last layer it would zero out the probability of a specific class completely and thus, hampering the performance of the model drastically.
- We for now have assumed the heuristic taken into consideration performs well in general. This is not the ideal case for all the models but pruning the lower magnitude weights would allow us to zero out certain weights while keeping the model performance fairly same.

PSEUDOCODE FOR PRUNING

```
1.  func prune_model(model, prune_percentage):
2.          Clone the model and name it pruned_model
3.          for layer in model.layers(except for the last one):
4.                  ranks[layer] = get_rank(weight[layer]) // Rank the weights element wise
5.                  threshold= ceil(max(ranks[l]) * prune_percentage) // Get the threshold value based on the value of prune percentage
6.                  ranks[layer][ranks[layer] <= threshold] = 0 // Assign rank elements to 0 that are less than or equal to the threshold
7.                  ranks[layer][ranks[layer] <= threshold] = 1  // Assign rank elements to 1 to those that are above.
8.                  weight[layer] = weight[layer]* ranks[layer] // Multiply weights array with ranks to zero out the lower ranked weights
9.          Copy the updated weights into cloned model
10.         return pruned_model
```

For future enhancements, we can use the improved heuristics as well as possibly use ideas like imposing L1 regularization for each of the connections so that the model while in the learning phase itself learns sparse connections.

3.3. Quantization Aware Training

Inspired by the paper, “Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference”, we decided to implement quantization aware training using different quantization schemes as opposed to the singular approach described in the paper. Additionally, quantization modules available on pytorch and tensor flow have been customized for INT8 operations alone. We wanted to evaluate its performance across varied precision levels by exploring the software aspect of quantization.

Quantization aware training models quantization during training itself and can generate better accuracies as compared to post-training quantization. Quantization-aware-training (QAT) is typically achieved by automatically inserting simulated quantization operations in the graph at training. In most quantization-aware training experiments, if weights are quantized to INT8, the input to the convolution/linear layer needs to be quantized and similarly, the output needs to be dequantized before moving to the next layer. However, in our experiments, we are only quantizing the weights to a fixed precision value which are represented as floating-point numbers due to hardware constraints. Thus, there is no need for us to quantize the input and dequantize the output. Thus, we insert simulated quantization operations in the graph at training time alone. Once the model is trained, only the quantized weights are used for inference.

At all times we maintain a copy of the full precision weights.

Forward Pass: For each batch during the training iteration, we update the weights as follows:

$$w_q = QuantizationTechnique\,(w_{fp})$$

w_q represents quantized weights, QuantizationTechnique refers to the quantization methods discussed in our 1st report such as stochastic quantization, rounding and mid-rise. w_fp refers to full precision weights. In the forward pass we thus use quantized weights.

Backward Propagation: The gradients are calculated with respect to quantized weights. This gradient is used to update the full precision weights. As such, in quantization aware training since full precision weights are updated with regards to quantized weights, they tend to generally outperform post training quantization techniques.

$$w_{fp} = w_{fp} - \frac{\partial L}{\partial w_q} . I_{w_q \in (w_{min}, w_{max})}$$

Straight through estimators are used to backpropagate through the quantization functions as using their gradients as-is would severely hinder the learning process.

Quantization Granularity: Only the weights of the model are quantized. We are thus quantizing all convolution and linear layers. Per-layer quantization is implemented. The bias has not been quantized as they account for only a tiny fraction of the parameters in a neural network. Furthermore, as each bias-vector entry is added to many output activations, any quantization error in the bias-vector tends to act as an overall bias. This must be avoided in order to preserve good end-to-end neural network accuracy. Additionally, the distribution of the bias vector is different from weights vector and quantization strategies for these could be explored in the future. To reduce model complexity and provide a basis of comparison with results generated in report 1, we are not quantizing the activation functions.

PSEUDOCODE FOR QUANTIZATION AWARE TRAINING:

```
1.  func quantization_aware_train_model(model):
2.          for epoch in range(epochs): // For each epoch in the training loop
3.                  for step in range(batch) : // For each batch in the training loop
4.                          model.train()
5.                          w_fp = model_params.clone() // Create a copy of full precision weights
6.                          w_q = quantize_weights(w_fp , precision, quant_method ) // Quantize weights per layer. Bias remains unquantized
7.                          model_params.copy(w_q) // Replace model weight parameters with quantized weights
8.                          loss = criterion(model(input), output) // Compute loss for quantized weight parameters
9.                          grad = loss.backwards() // Compute gradient i.e. loss with respect to quantized weights
10.                         grad_ste = grad * ste_approximation // Update gradient as per straight through estimator approximations
11.                         w_fp = w_fp - learning_rate * grad_ste //Update the full precision weights
12.                         model_params.copy(w_fp) //Replace model parameters with updated learnt weights
13.         return qat_model
```

Please refer to Appendix section 8.3 Fig 1, for further details. In addition to implementing this algorithm from scratch for multiple precision values and multiple quantization techniques, in parallel we also tried to use built-in pytorch functionalities. The objective was to compare the results from these two processes. In pytorch's quantization aware training module, the parameters can only be quantized to INT8 and certain functions are hardware dependent. The code base was running successfully for DNN based classification models on Churn Dataset on CPU. However, when implementing CNN models, quantized Conv2d required specialized backends such as QuantizedCPU and as such we couldn't proceed further with our experiment.

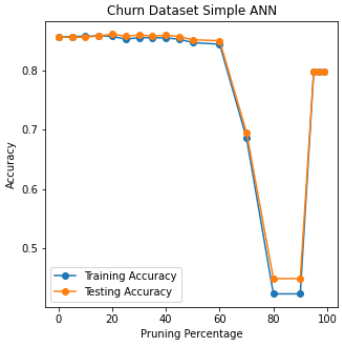
4. Results

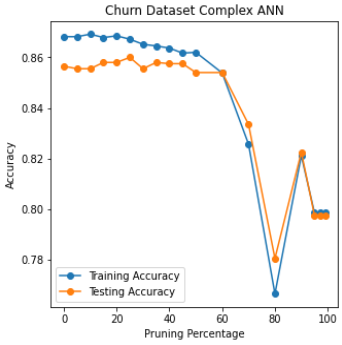
4.1. Pruning

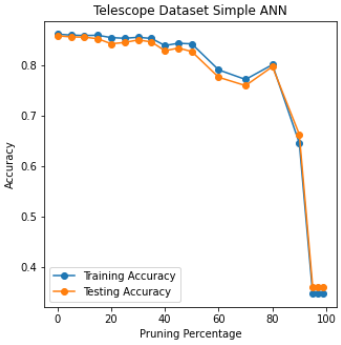
We ran all the full-precision models for classification and regression datasets mentioned in the report 1 across different pruning percentages and recorded model's performance for each of those. Following plots show the model performance (accuracy in case of classification and loss in case of regression tasks) for different datasets at different pruning levels.

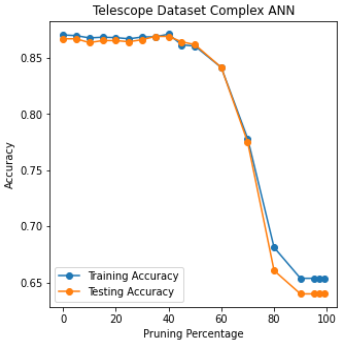
4.1.1. Results for Pruning

- **ANN-based Classification**

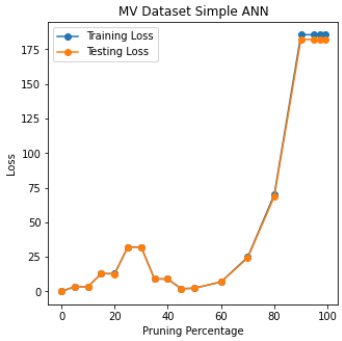


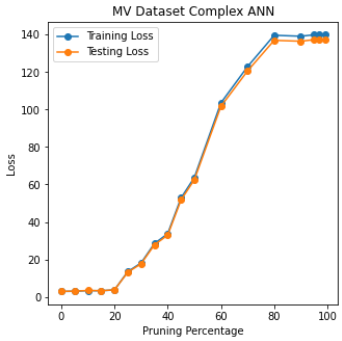


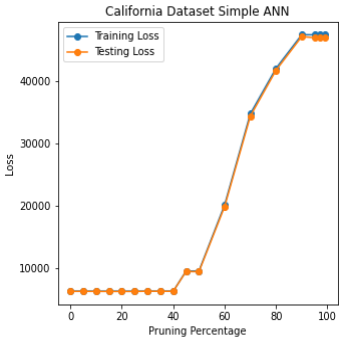


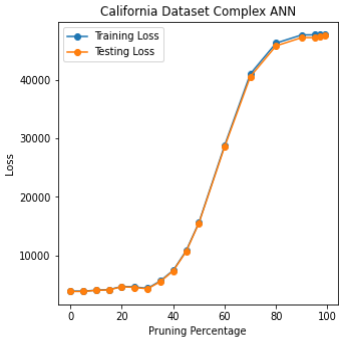


- **ANN-based Regression**

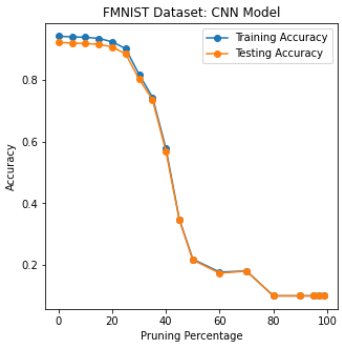


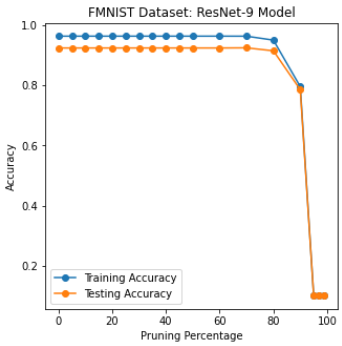


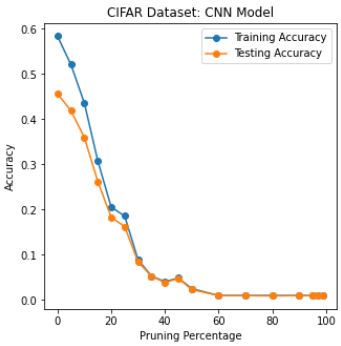


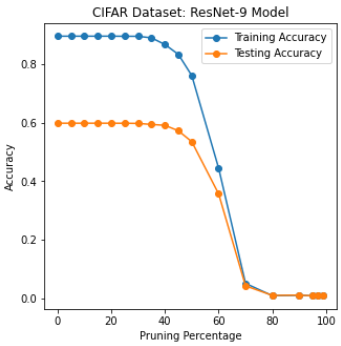


- **Convolutional Neural Networks**



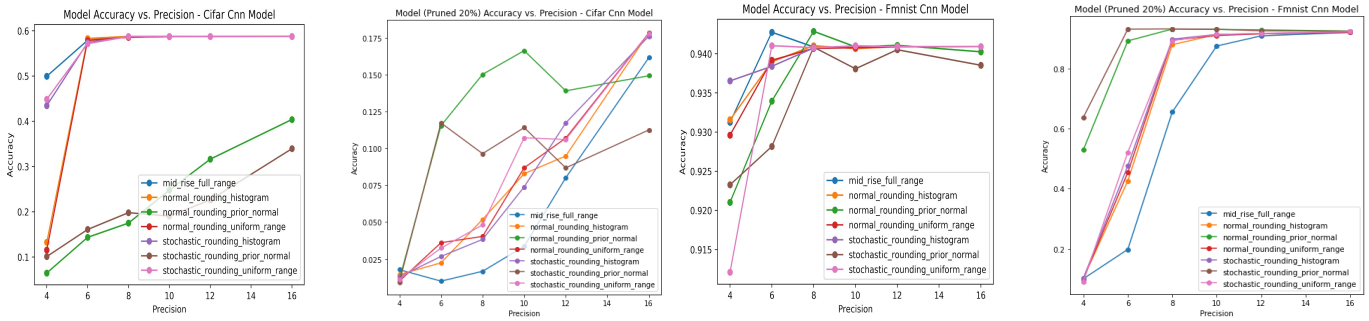






4.1.2. Pruning on Quantized Weights

Based on the results obtained from pruning we observe that in most of the cases 20% pruning doesn't result in a major accuracy drop and hence, we combine the quantization along with 20% pruning for each model. Here, we first quantize the model and after that prune it by 20%. This allows us to have a 20% reduction in an already compressed network obtained through pruning. We are not guaranteed to have zero levels matched in all the output networks obtained after quantization as not all quantization methods maintain the same zero point in input and output. Thus, to have the advantages of having zeroed weights in the network we apply pruning after quantizing the networks. Results for CNN based tasks are as shown below:



The precision on X-axis represents values in bits. N bit precision implies having 2^N unique possible values.

4.1.3. Key Takeaways

From the plots shown in the pruning section, it is evident that in a more complex model we have more chances of having redundant weights thus, allowing us to have a sparser network. The results also show that the plots for complex models are smoother than the simpler model again supporting the fact that in complex networks we have redundant weights which in the case of simpler models won't be a case.

Another key takeaway from the results would be that the performance of the network after pruning depends on how well the model without pruning was learnt. If the model has overfitted without pruning we can use pruning like a regularization technique which would help us learn a more generalizable model. While when the model without pruning is already performing suboptimal, pruning would affect its performance a lot as all the weights in that network state aren't finalized and hence, each of them play an important role towards the overall performance of the network.

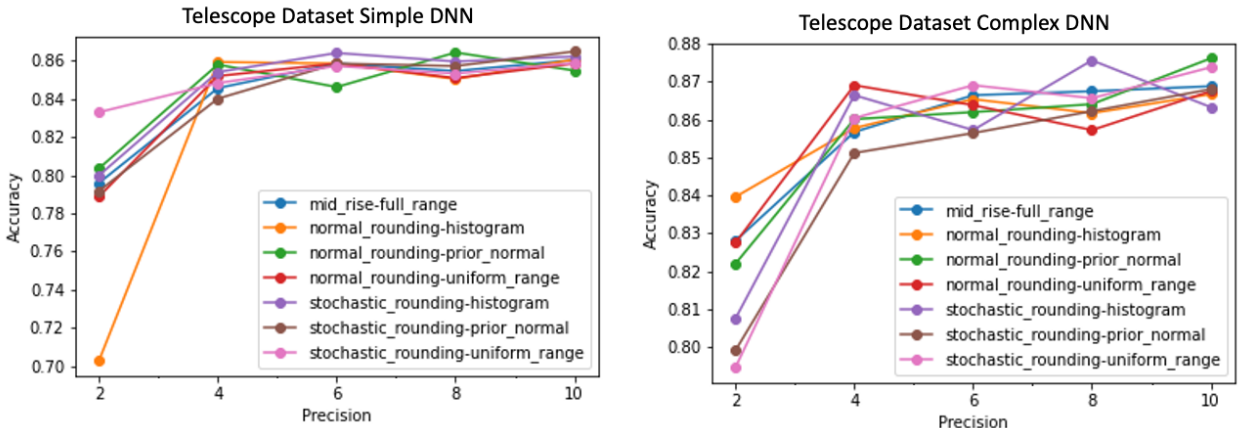
From the results of pruning and pruning combined with quantization it is evident that we need to choose the pruning percentage wisely otherwise it hampers the overall performance of quantized models quite strongly. This is reflected in the performance of CNN for the CIFAR dataset. From the pruning task independently it is clear that when we apply 20% pruning for CIFAR dataset, the performance goes sub-optimal and thus, applying 20% pruning after quantizing the models results into larger reduction in the performance of the model.

4.2. Quantization Aware Training

In this section we would be evaluating the results of quantization aware ANN based classification models at varied precision levels. We would also be comparing the results with those of simple post quantization.

Since our current computation capabilities are limited to Google Colab, we could not evaluate this technique for CNN models. However, in our previous experimentation we had observed that accuracy dropped significantly for CNN models with larger numbers of parameters at lower precision. This was evident as it resulted in significant information loss. Additionally, accuracy was lower at low precision values for datasets with a huge number of classes. The effects of QAT would thus be more impactful on CNN models. We are currently working on optimizing our code further.

4.2.1. ANN-based Classification



*The precision on X-axis represents values in bits. N bit precision implies having 2^N unique possible values.

We observe that there is not much variability in accuracy across different quantization techniques in quantization aware training as opposed to our previous report. We had earlier observed that stochastic rounding-prior normal and normal rounding-prior normal both performed relatively poorly at all precision levels. In QAT, they all seem to be performing well on this dataset. This seems logical due to the way QAT updates model parameters. In particular, at precision level 4 for the simple model we observe that accuracy ranges between 0.83-0.85 as opposed to our previous report on post-quantization training where the lowest accuracy at 4-bit precision was 0.80. The accuracy drop at 2 bit precision is more pronounced in the simple model as compared to the complex model. Overall the results from QAT seem to be similar to our previous post-quantization results. Since, the full precision model is well fitted on the data and the post quantization methods also perform satisfactorily, investing in QAT method for quantization for marginal improvements is less desirable.

5. Next Steps

- Multipoint Quantization: Experiment on CIFAR 100 and FMNIST datasets for vanilla CNN, ResNet9, ResNet50, and VGG16 model at precision levels of 8, 6, 4, and 2.
- Quantization-Aware Training: Experiment on CIFAR 100 and FMNIST datasets for vanilla CNN, ResNet9, ResNet50, and VGG16 model at precision levels of 8, 6, 4, and 2.
- Post-Training Quantization + Pruning: Compare the performance of post-training quantization and post-training quantization + pruning.
- Model Size: Explore ways to quantify the model size in order to compare level of model compression for various quantization techniques explored thus far.

- Experiment Design: To find statistical evidence for the results described in terms of accuracy and model size.

6. Individual Contribution

- **Pritam Biswas** - Researched on Quantization Aware Training for neural networks. Developed a weight-only based QAT for CNN and dense neural network. Implemented a straight through estimator for updating gradients in the training process. Ran simulations of the dense neural nets, with precisions from 2,4,6,8,10.
- **Neelam Patodia -** Focussed on quantization aware training : literature survey on the techniques used, developed the utility code, POC on straight through estimators and implementation of QAT scheme, experimented with built-in quantization aware modules on Pytorch
- **Mohit Chander Gulla -** Literature survey of various post-training quantization methods. Worked on implementation of multipoint post-training quantization - function to create quantization set, store input activatations of each layer on calibration data, and overall model-level implementation of multipoint quantization.
- **Kumari Nishu -** Studied post-training quantization methods - multipoint, RL based post training quantization.. Worked on implementation of multipoint post-training quantization - Quantization algorithm, Working on running experiments for various datasets and various CNN models.
- **Prasham Dhaneshbhai Sheth -** Did literature survey of various post-training quantization methods focusing on combination of pruning and quantization techniques. Worked on implementation of pruning, experimenting with pruning for different models, and combining the tasks of pruning and post-quantization designed previously.

7. References

[1] Xingchao Liu, Mao Ye, Dengyong Zhou, & Qiang Liu. (2020). Post-training Quantization with Multiple Points: Mixed Precision without Mixed Precision.

[2] Shaokai Ye and Tianyun Zhang and Kaiqi Zhang and Jiayu Li and Jiaming Xie and Yun Liang and Sijia Liu and Xue Lin and Yanzhi Wang (2018). A Unified Framework of DNN Weight Pruning and Weight Clustering/Quantization Using ADMM CoRR, abs/1811.01907.

[3] Boyd, S., Parikh, N., Chu, E., Peleato, B., & Eckstein, J. (2011). Distributed Optimization and Statistical Learning via the Alternating Direction Method of MultipliersFound. Trends Mach. Learn., 3(1), 1–122.

[4] Benoit Jacob and Skirmantas Kligys and Bo Chen and Menglong Zhu and Matthew Tang and Andrew G. Howard and Hartwig Adam and Dmitry Kalenichenko (2017). Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference CoRR, abs/1712.05877

[5] Kuan Wang and Zhijian Liu and Yujun Lin and Ji Lin and Song Han (2018). HAQ: Hardware-Aware Automated Quantization CoRR, abs/1811.08886.

[6] Song Han and Jeff Pool and John Tran and William J. Dally (2015). Learning both Weights and Connections for Efficient Neural NetworksCoRR, abs/1506.02626.

[7] https://pytorch.org/docs/stable/nn.html#utilities

[8] https://www.tensorflow.org/model_optimization/guide/pruning

[9] Raghuraman Krishnamoorthi (2018). Quantizing deep convolutional networks for efficient inference: A whitepaper CoRR, abs/1806.08342.

8. Appendix

8.1. Methodology of Post-Training Quantization with Multiple Points: Mixed Precision without Mixed Precision

The concept of mixed-precision is a recent innovation to boost the performance of quantized neural networks. The idea is to assign more bits to important layers, and fewer bits to unimportant layers to better control the overall quantization error and balance the accuracy and cost more efficiently. However, as this requires specialized hardware, it is not very feasible. Multipoint quantization achieves the same flexibility as mixed precision but using only a single-precision level. The process can be logically broken down as building a quantization set, performing multipoint quantization for each layer of a neural network (fully-connected or convolutional), and an optimization procedure to reduce approximation error.

The quantization set, Q, is constructed using a uniform grid on $[-1,\ 1]$ with increment ε_b between the elements where $K > 0$ is a scaling factor that controls the length of Q and B specifies the center of Q, as defined below:

$$\mathcal{Q} = K \times [-1 : \epsilon_b : 1] \ + \ B, \quad \epsilon_b := \frac{1}{2^{b-1} - 1}$$

The multipoint quantization algorithm approximates real-valued weight $w = (w_1,\ w_2,\ \ldots,\ w_d) \in R^d$ with a weighted sum of a set of low precision weight vectors as:

$$\tilde{\boldsymbol{w}} = \sum_{i=1}^{n} a_i \tilde{\boldsymbol{w}}_i$$

where $a_i \in R$ and $\tilde{w} \in Q^d$ for all $i = 1,\ \ldots,\ n$. A naive quantization approximates a weight by choosing the nearest grid points, while multipoint quantization approximates it with the nearest point on the linear segments. Given a fixed n, we want to find the optimal $\{a_i^*,\ w_i^*\}_{i=1\ldots n}$ that minimizes l_2-norm between the real-valued weight and the weighted sum:

$$\{a_i^*, \tilde{\boldsymbol{w}}_i{}^*\}_{i=1}^n = \underset{\{a_i, \tilde{\boldsymbol{w}}_i\}_{i=1}^n}{\arg\min} \left|\left| \boldsymbol{w} - \sum_{i=1}^{n} a_i \tilde{\boldsymbol{w}}_i \right|\right|$$

As n increases, the dimension of the approximation set increases. Intuitively, the nearest distance from an arbitrary point to the approximation set decreases exponentially. We apply the above multipoint quantization iteratively layer by layer to the entire neural network. We can significantly reduce the quantization error of a layer or a channel using this approach.

8.2. Methodology of Unified Framework of DNN Weight Pruning and Weight Clustering / Quantization Using ADMM

In their work to unify the pruning and quantization techniques, they formulate the problem of finding optimal weights as a constraint satisfaction problem as described below:

For an N-layer Neural network with weights and bias for layer i denoted as W_i and b_i, the goal is to find W_i and b_i that minimize the loss function f. The constraints here would be the need of W_i to belong in sets S_i and S_i'.

$$\begin{aligned} \underset{\{\mathbf{W}_i\},\{\mathbf{b}_i\}}{\text{minimize}} \quad & f\big(\{\mathbf{W}_i\}_{i=1}^N, \{\mathbf{b}_i\}_{i=1}^N\big), \\ \text{subject to} \quad & \mathbf{W}_i \in \mathbf{S}_i,\ \mathbf{W}_i \in \mathbf{S}_i',\ i = 1, \ldots, N. \end{aligned}$$

- The set Si reflects the constraint for the weight pruning problem:
  S_i= {the number of nonzero elements is less than or equal to αi}, where αi is the desired number of weights after pruning in the i-th layer.
- The set S_i' reflects the constraint for the weight pruning problem:
  S_i' = {W_i | the weights in W_i only take values from the set {$Q_1, Q_2, \cdots, Q_{Mi}$}}. Here the Q values are quantization levels. Besides, $M_i = 2^n$, and n is the number of bits we use for weight clustering or quantization.

They solve this constraint satisfaction problem using ADMM which decomposed the constraint problem into two subproblems that can be solved separately and efficiently. To apply ADMM they introduce ine indicator functions to incorporate the combinatorial constraints into the objective function.

$$g_i(\mathbf{W}_i) = \begin{cases} 0 & \text{if } \mathbf{W}_i \in \mathbf{S}_i, \\ +\infty & \text{otherwise,} \end{cases}$$

$$h_i(\mathbf{W}_i) = \begin{cases} 0 & \text{if } \mathbf{W}_i \in \mathbf{S}_i', \\ +\infty & \text{otherwise,} \end{cases}$$

They further incorporate auxiliary variables Z_i and Y_i, and rewrite the original problem as:

$$\begin{aligned} \underset{\{\mathbf{W}_i\},\{\mathbf{b}_i\}}{\text{minimize}} \quad & f\big(\{\mathbf{W}_i\}_{i=1}^N, \{\mathbf{b}_i\}_{i=1}^N\big) + \sum_{i=1}^N g_i(\mathbf{Z}_i) + \sum_{i=1}^N h_i(\mathbf{Y}_i), \\ \text{subject to} \quad & \mathbf{W}_i = \mathbf{Z}_i,\ \mathbf{W}_i = \mathbf{Y}_i,\ i = 1, \ldots, N. \end{aligned}$$

Through ADMM they finally decompose the above stated problem into 3 subproblems as below. The overall problem of weight pruning and clustering/quantization is solved through solving the subproblems.

$$\underset{\{\mathbf{W}_i\},\{\mathbf{b}_i\}}{\text{minimize}} \quad f\left(\{\mathbf{W}_i\}_{i=1}^N,\{\mathbf{b}_i\}_{i=1}^N\right)+\sum_{i=1}^N \frac{\rho_i}{2}\|\mathbf{W}_i-\mathbf{Z}_i^k+\mathbf{U}_i^k\|_F^2+\sum_{i=1}^N \frac{\rho_i}{2}\|\mathbf{W}_i-\mathbf{Y}_i^k+\mathbf{V}_i^k\|_F^2$$

$$\underset{\{\mathbf{Z}_i\}}{\text{minimize}} \quad \sum_{i=1}^N g_i(\mathbf{Z}_i)+\sum_{i=1}^N \frac{\rho_i}{2}\|\mathbf{W}_i^{k+1}-\mathbf{Z}_i+\mathbf{U}_i^k\|_F^2$$

$$\underset{\{\mathbf{Z}_i\}}{\text{minimize}} \quad \sum_{i=1}^N h_i(\mathbf{Y}_i)+\sum_{i=1}^N \frac{\rho_i}{2}\|\mathbf{W}_i^{k+1}-\mathbf{Y}_i+\mathbf{V}_i^k\|_F^2$$

To simplify the complexity of the above shown problem, they further split it into two wherein they apply pruning first followed by the quantization/clustering technique both of which are solved using the ADMM technique.

8.3. Methodology of Quantization and Training of Neural Networks for Efficient Integer Arithmetic Only Inference

The quantization scheme is implemented using integer-only-arithmetic during inference and floating-point arithmetic during training, with both implementations maintaining a high degree of correspondence with each other.

The quantization scheme is: $r = S(q - Z)$

For, r denotes real numbers and q denotes integers. For *8-bit quantization*, q is quantized as an 8-bit integer. The constants S and Z are the quantization parameters and denotes scale and zero-point respectively. This quantization scheme uses a single set of quantization parameters for all values within each activa- tions array and within each weights array; separate arrays use separate quantization parameters. The bias vectors are quantized as 32 bit integers as they account for only a tiny fraction of the parameters in the neural network.

In order to overcome the challenges of simple post-training quantization related to outlier weights and large differences in range of weights, the authors propose a simulated quantization effect in the forward pass. Here, weights are quantized before they are passed to the convolution layer. If batch normalization is used, it is folded into the weights before quantization. Activation functions are quantized at points where they would be during inference. Weight quantization ("wt quant") and activation quantization ("act quant") nodes are thus inserted into the computation graph to simulate the effects of quantization of these parameters. Backpropagation happens as usual i.e. weights and biases are stored in floating points. This has been illustrated in the figure below.

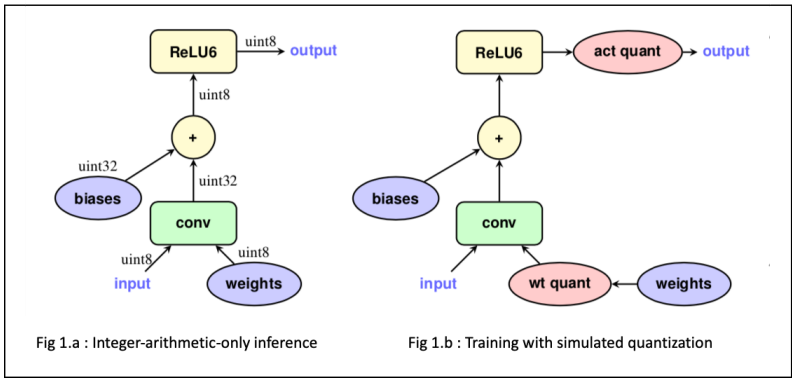


Fig 2: Integer-arithmetic-only quantization

8.4. Methodology of HAQ: Hardware-Aware Automated Quantization with Mixed Precision

The approach of the paper is motivated by the reasoning that different layers of DNN have different redundancy and have different arithmetic intensity (OPs/byte) on the hardware, which advocates for using mixed precision for different layers. In this reinforcement learning based approach, the RL agent learns to output the optimal quantization policy for the given DNN model and the hardware configuration based on the resource constraints (latency, power, model size) and the validation accuracy of the quantized model ran on the specified hardware as feedback

The RL training process to find the optimal policy can be summarized in the below flow chart. The RL agent takes the initial bidwidth assignment and the information regarding all layers as input at time t and then outputs the bit width assignment at next time t as an action. Post that, the newly created bitwidth assignment is used to quantize the model and get validation accuracy and also checked if the quantized model meets the strict resource restriction, both are then passed as feedback to the RL agent. In the below section, we have briefly described the workflow of the RL agent.

Notation:

- **k** denotes layer index, if we have L layers in our DNN then k will be [1,2,3,...,L]
- Observation $\mathbf{O_k}$ for a convolution layer will include (layer index k, number of input channel, number of output channel, kernel size, stride, input feature map size etc.)
- We use **t** to denote the output policy by the RL agent at different time indexes.
- {a1,a2,..aL} denotes the optimal bit width for each layer i.e a1=4 implies that the layer 1 should be quantized with 4bits.

Training an RL agent for a DNN architecture and a hardware configuration

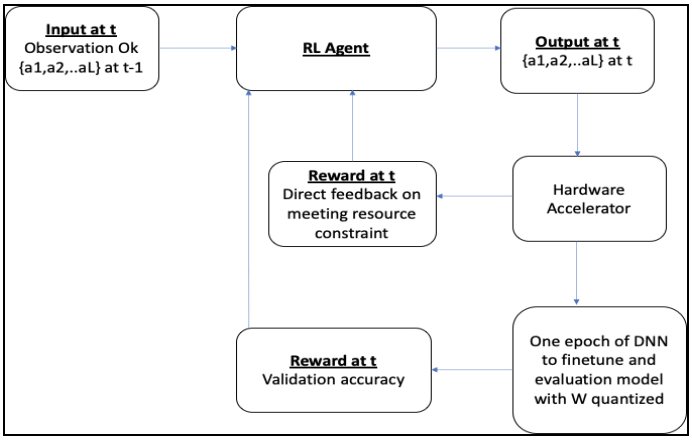


Fig 3: Optimal Quantization Policy Using RL agent

Hence, this provides an automated framework for post training quantization which does not require any domain experts and rule based heuristics to decide the quantization bit width to be used for different layers. The paper has shown that the optimal policies on different hardware architectures (edge and cloud architectures) under different resource constraints (i.e.,latency, energy and model size) are drastically different.