ECE 661: Homework #4

Elisa Chen (Duke ID: eyc11)

2023-11-04

Q1 True / False Questions

Problem 1.1: **True** - weight pruning attempts to reduce the number of parameters in the model by setting a portion of the weights to 0 or pruning neurons from the network while weight quantization will map the the weights to a lower precision therefore reducing the memory footprint of the model. These two techniques do not interfere with one another and can be applied to the model simultaneously leading to greater model compression benefits.

Problem 1.2: **True** - typically, iterative pruning will lead to non-structured sparse weight matrices as many of the weights will be zeroed. Depending on the distribution, the gains on the speedup might be marginal to non-existent on traditional platforms like GPU therefore NOT affecting the inference latency. Specialized hardware might need to be used to gain any efficiencies in the inference latency for pruned models.

Problem 1.3: **False** - I would argue that if quantization step was skipped in the deep compression pipeline, Huffman coding process wouldn't be nearly as effective as it would be with quantization in place. Huffman coding is particularly effective when there are few dominant weight values, which typically only occurs after quantization stage. Pruning will decrease the number of parameters, but doesn't necessarily reduce the distinct number of weights.

Problem 1.4: **False** - It is false to assume that all sparsity-inducing regularizers will lead to zero values in the weight elements. There are several factors influencing the final number of non-zero weights in the network including the strength of the regularization (often denoted as λ) in the loss function. Depending on the value of lambda, the number of non-zero weights will vary greatly and if lambda is too small, there might be no weights with exact zero values. Additionally, you might still want to apply pruning methods after regularization especially if it's crucial to reach a certain model compression size that wasn't achieved through regularization.

Problem 1.5: **True** - the l1 regularization term is effective in introducing sparsity in the weight matrix. However, it can be quite biased as the absolute value of large elements always shrinks by the regularization term leading to loss of variance. Soft thresholding operator can lead to better results as it allows for smoother convergence of the overall objective.

Problem 1.6: **False** - It is true that Group Lasso as a method can lead to more structured sparsity on DNNs, which can result in speedups on GPUs. However, the all-zero groups are formed when we apply L-1 regularization to the L-2 norms of all the groups, not the other way around.

Problem 1.7: **True** - The proximal term is designed to encourage or pull the weight parameters toward 0 and the inclusion of the proximal term allows for smoother convergence of the overall objective during optimization.

Problem 1.8: **True** - Models equipped with early exits are designed to allow some inputs to be computed with only part of the model, enabling them to make predictions or classifications at intermediate stages of the model's architecture. This can reduce overfitting and overthinking by acting as regularization to provide intermediate supervision and prevent models from becoming overly complex.

Problem 1.9: **False** - When implementing quantization-aware training with STE, forward propagation is performed using quantized weights, but backpropagation is performed using full-precision weights, because the gradient needs to be performed with full precision to update the weight set.

Problem 1.10: **True** - Mixed-precision quantization can provide a better size and latency-accuracy tradeoff than fixed quantization. In other words, for a similar sized model, the mixed-precision quantization can yield a higher accuracy and this was demonstrated with an experiment performed with MobileNets introduced in Lecture 15.

Lab 1: Sparse optimization of linear models

a): Please see below for the formulaic derivation of the weight W^{k+1} after step k + 1 of full-batch gradient descent with $X_i, y_i, i \in 1, 2, 3$:

The general form for updating W^{k+1}:

$$W^{k+1} = W^k - \mu \frac{dL}{dW^k}$$

where

$$\frac{dL}{dW^k} = \frac{d(\Sigma_i (X_i W^k - y_i)^2)}{dW^k} = 2\Sigma_i (X_i W^k - y_i) X_i$$

and putting everything together we obtain the formula for weight W^{k+1}:

$$W^{k+1} = W^k - \mu * 2\Sigma_i^3 (X_i W^k - y_i) X_i$$

for $i \in 1, 2, 3$.

b): We can observe that by using vanilla full-batch gradient descent, we converge quite quickly for all weights after ~25 steps. Furthermore, we do not see any significant changes in the log loss after ~170 steps increasing our confidence in the fact that we've converged to an optimal solution using gradient descent. However, it's evident from the results that we do NOT converge to a sparse solution where we would have 2 or less non-zero weights. Even if one could argue that the third and fourth weight elements are close enough to 0 to be removed from the weight matrix, the magnitude of the first, second and the fifth weight elements are too significant to be removed still leaving us with three non-zero weights. Please see Figure 1 for more details.

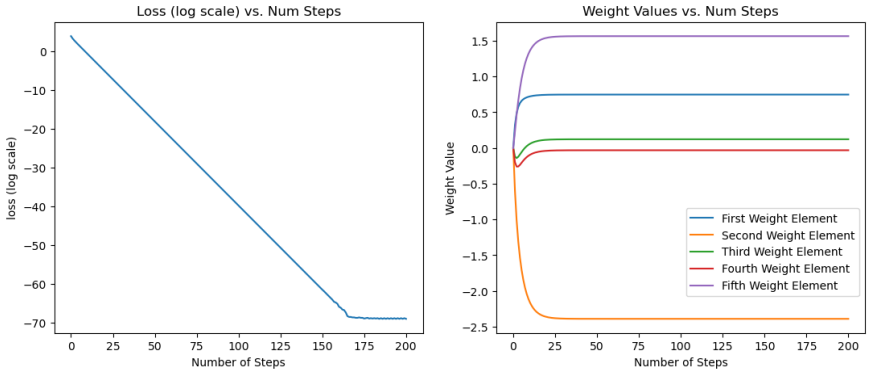

Figure 1: Log MSE Loss (Left) and Weight Values During Training With Vanilla GD (Right)

c): Using projected gradient descent to enforce the sparse constraint of $||W||_0 \leq 2$ will converge to an optimal solution AND a sparse solution. We can observe that we're still converging to an optimal solution

after about ~170 steps just like before, but this time, we are also achieving a sparse solution of 2 non-zero weights (second and fifth weight elements) using projected gradient descent to enforce the sparse constraint. Please see `Lab2-LM.ipynb` part `c)` for more details on coding implementation and and Figure 2 for more details on the results.

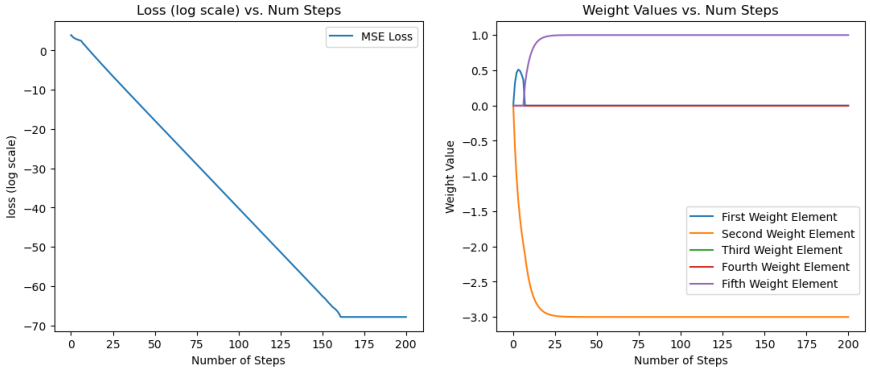

Figure 2: Log MSE Loss (Left) and Weight Values During Training (Right) With PGD

d): After changing the minimization objective to include the regularization term $\lambda||W||_1$, we observe that we converge a lot sooner to an optimal solution regardless of the strength (λ) of the regularization. From Figure 3 we can observe that all the L1 loss functions converge after ~25 steps. The higher the strength of regularization, the higher the loss value for the converged solution, which aligns with our intuition of introducing an additional constraint in the loss function. Similarly, the higher the value of λ, the sooner some weights move towards zero. When $\lambda = 0.2$, we observe that we do not reach a sparse solution with 2 or less non-zero weights. This makes sense intuitively as the vanilla loss function influences the overall loss more than the regularization term resulting in a similar outcome as we did in part b). While we do reach a sparse solution with other lambda values, we also notice that there's a good degree of noise introduced in the weights that are guided towards zero which seems to increase as the value of lambda increases. Please refer to Figure 4 for more details. The code implementation for L1-Loss can be found in `Lab2-LM.ipynb` part `d)`.

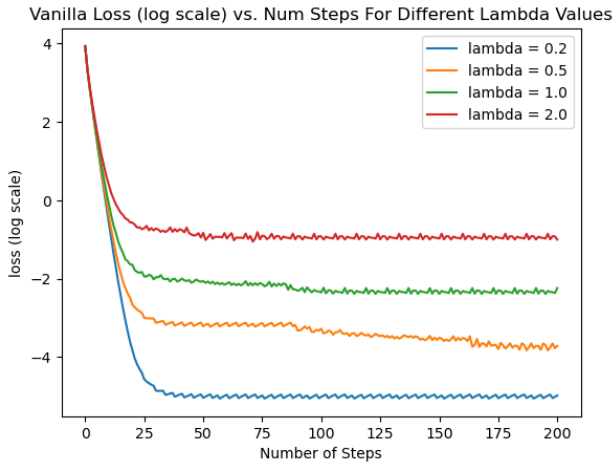

Figure 3: Log L1 Loss

e): As we know, optimizing $L + \lambda||W||_1$ using gradient descent with learning rate μ should correspond to proximal gradient update with a threshold of $\mu\lambda$, which means that we should be expecting similar convergence performance as we obtained in part d). The loss functions for each threshold $\mu\lambda$ are very similar to what we observe in part d). The higher the threshold value, the higher the loss value for the converged

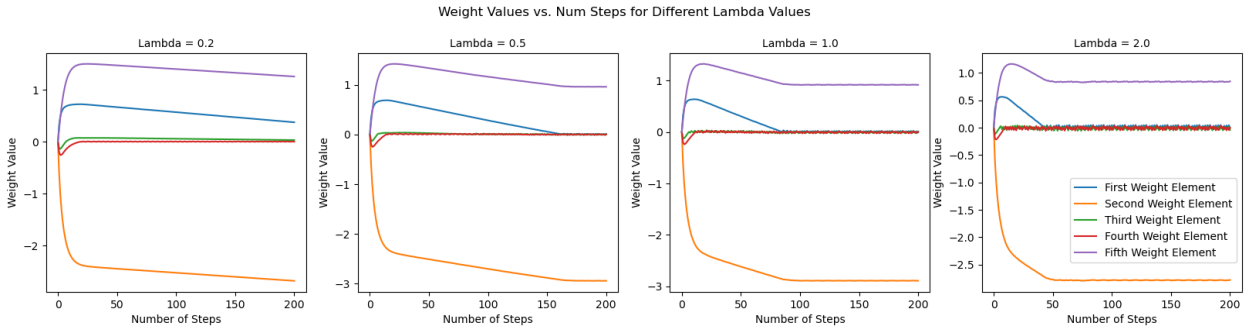

Figure 4: Weight Values During Training With L1 Loss

solution. From Figure 6 we can see that the weights converge to a sparse solution for all but the smallest threshold just like we observed in part d). The convergence speed is similar to what we obtained in part d) for each threshold compared to its corresponding lambda value. However, unlike with L1 optimization in part d), proximal gradient update allows for a smoother convergence to the overall objective, which is reflected by the lack of noise in the loss function and weight values that were present in part d). The code implementation for Proximal Gradient Update can be found in `Lab2-LM.ipynb` part `e)`.

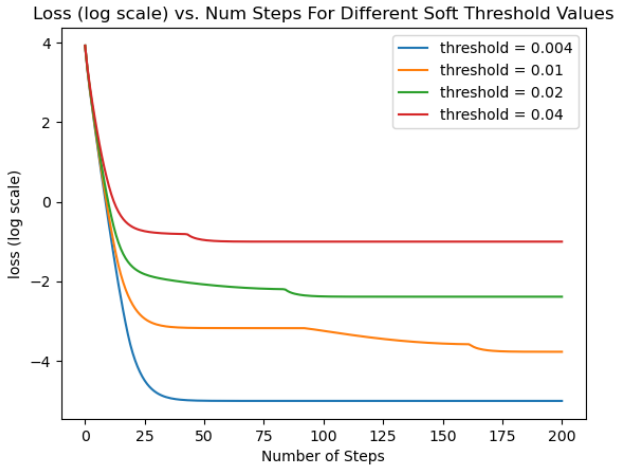

Figure 5: Log Loss With Proximal Gradient Update

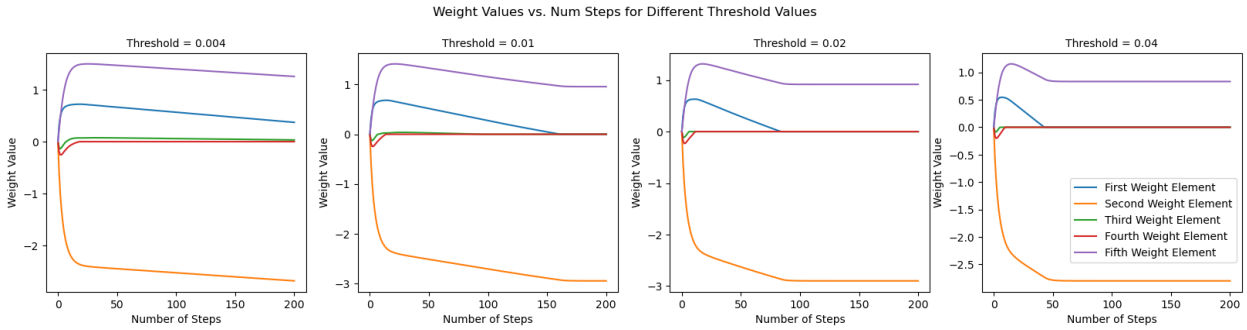

Figure 6: Weight Values During Training With Proximal Gradient Update

f): When applying the trimmed regularizer or the proximal gradient update on the three smallest elements each time, the model is able to achieve similar loss levels as in part a). We observe that all loss functions, except when the threshold is 0.02, converge before 200 steps reaching loss levels significantly lower than those when we applied l1 regularization shown in Figure 7. It also seems like the higher the threshold, the sooner the model is able to converge. This is also reflected in Figure 8 where we observe weights converging sooner as the threshold and the lambda increases. We obtain a sparse solution in all four cases. When comparing the behavior of early steps between the Trimmed l_1 and the iterative pruning, we see that in both training methods, the loss decreases exponentially at first before decaying at a linear rate (at a log scale). However, the value of the threshold seems to control the velocity at which the loss initially decreases exponentially. When the threshold is 0.2, the loss decays faster than it does for iterative pruning, but for a smaller threshold, iterative pruning results in a faster decay. Please see figures 9 and 10 for more details. The code implementation for Trimmed L1-Loss can be found in `Lab2-LM.ipynb` part `f)`.

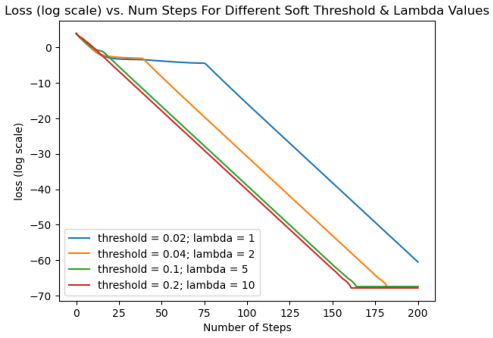

Figure 7: Log Loss With Proximal Gradient Update - Trimmed l1

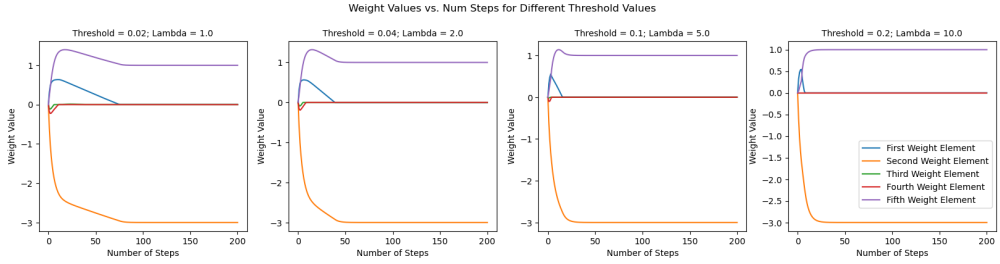

Figure 8: Weight Values During Training With Trimmed l1

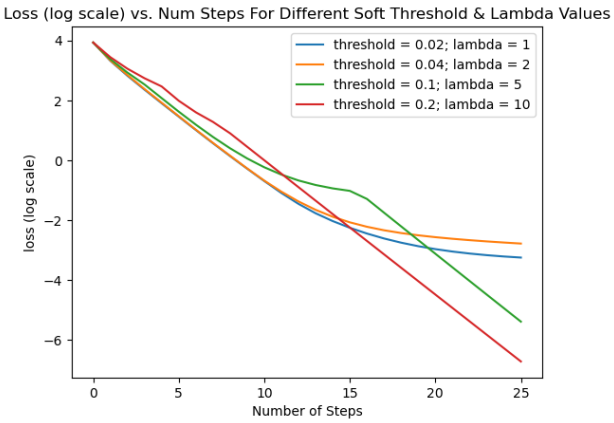

Figure 9: First 20 steps with Trimmed L1

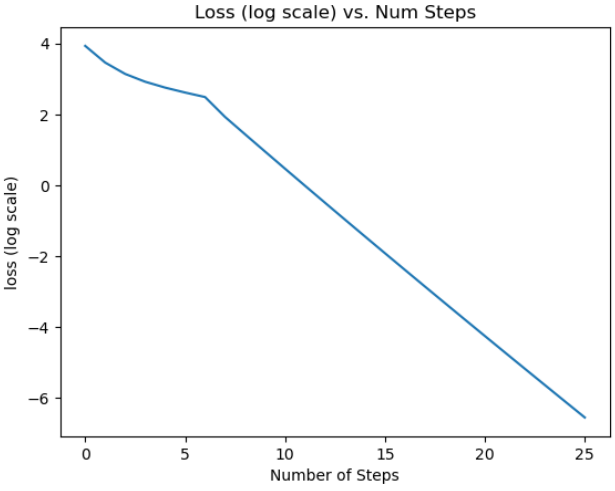

Figure 10: First 20 steps of Iterative Pruning

Lab 2: Pruning ResNet-20 Model

a): The test accuracy of the floating-point pre-trained model is: 91.51%.

b): Please refer to the `hw4.ipynb` notebook for more details on the `prune_by_percentage` function implementation. The function returns a mask indicating which weights should be zeroed for the given layer. The test accuracy for each q = 0.3, 0.5, 0.7 are provided below:

q	Test Accuracy
0.3	90.28%
0.5	82.10%
0.7	42.04%

Setting q = 0.3 doesn't seem to significantly worsen the test accuracy, but increasing the % of pruned weights to 0.5 and 0.7 will significantly impact the performance of the model leading to test accuracy of 82% and 42% respectively.

c): Please refer to the `hw4.ipynb` notebook for more details on the `finetune_after_prune` function implementation and the checks to ensure that the finetuned model preserves the required sparsity. In the code, it was ensured that the pruned away elements in previous step were kept as 0 throughout the finetuning process. After setting q=0.7 for 20 epochs, a **best test accuracy of 89.46%** was achieved. This is a significant improvement from part b) in which the baseline accuracy was 42% without the finetuning process.

d): Please refer to the `hw4.ipynb` notebook part Lab2 (d) for more details on the iterative pruning before finetuning implementation. The pruning percentage was increased linearly for 10 out of 20 epochs until it reached 70%. We can observe that by iteratively pruning the weights, we gain a marginal **improvement in test accuracy of 0.01%**. The best test accuracy with iterative pruning was 89.47%. The difference is too insignificant to conclude that the iterative pruning improved the training process.

e): Please refer to the `hw4.ipynb` notebook part Lab2 (e) for more details on the global iterative pruning implementation. Previously, the sparsity during each layer was constant determined by `q`, but for global pruning, we see that the sparsity for each layer varies a lot as shown in Figure 11. It seems that most of the pruning was done during the later convolutional layers of the model and the least pruning was performed during the fully-connected layer. The total sparsity level of the model was `q` as we'd expect. The global iterative pruning achieved **a best test accuracy of 90.06%**, which was the best accuracy obtained out of all the other methods we've experimented with in Lab 2.

```
Sparsity of head_conv.0.conv: 0.24305555555555555
Sparsity of body_op.0.conv1.0.conv: 0.5512152777777778
Sparsity of body_op.0.conv2.0.conv: 0.5269097222222222
Sparsity of body_op.1.conv1.0.conv: 0.5199652777777778
Sparsity of body_op.1.conv2.0.conv: 0.5529513888888888
Sparsity of body_op.2.conv1.0.conv: 0.5186631944444444
Sparsity of body_op.2.conv2.0.conv: 0.5655381944444444
Sparsity of body_op.3.conv1.0.conv: 0.5251736111111112
Sparsity of body_op.3.conv2.0.conv: 0.5832248263888888
Sparsity of body_op.4.conv1.0.conv: 0.6157769097222222
Sparsity of body_op.4.conv2.0.conv: 0.6769748263888888
Sparsity of body_op.5.conv1.0.conv: 0.6119791666666666
Sparsity of body_op.5.conv2.0.conv: 0.7030164930555556
Sparsity of body_op.6.conv1.0.conv: 0.6143120659722222
Sparsity of body_op.6.conv2.0.conv: 0.6512586805555556
Sparsity of body_op.7.conv1.0.conv: 0.66259765625
Sparsity of body_op.7.conv2.0.conv: 0.7189127604166666
Sparsity of body_op.8.conv1.0.conv: 0.7477213541666666
Sparsity of body_op.8.conv2.0.conv: 0.9371744791666666
Sparsity of final_fc.linear: 0.1203125
Total sparsity of: 0.6999992546657922
Files already downloaded and verified
Test Loss=0.3151, Test accuracy=0.9006
```

Figure 11: Sparsity At Each Layer With Global Iterative Pruning

Lab 3: Fixed-point quantization and finetuning

a): Please see the coding implementation for the `STE` forward function in the `FP_layers.py` notebook. The code for the `STE` class is also included in Figure 12. In this class, we're scaling the parameters before passing them through a linear k-bit quantizer.

```
class STE(torch.autograd.Function):
    @staticmethod
    def forward(ctx, w, bit, symmetric=False):
        '''
        symmetric: True for symmetric quantization, False for asymmetric quantization
        '''
        if bit is None:
            wq = w
        elif bit==0:
            wq = w*0
        else:
            # Build a mask to record position of zero weights

            weight_mask = torch.where(w != 0, 1, 0)

            # Lab3 (a), Your code here:
            if symmetric == False:
                # Compute alpha (scale) for dynamic scaling - QUESTION: do we perform global or local scaling?

                w_copy_min = w.clone() # for calculating minimum value
                w_copy_min[weight_mask == 0] = w.max() # to ensure that these values will not be picked as the minimum

                alpha = w.max() - w_copy_min.min() # only applied to non-zero weights. - take it for the entire weight matrix.
                # Compute beta (bias) for dynamic scaling
                beta = w_copy_min.min()
                # Scale w with alpha and beta so that all elements in ws are between 0 and 1
                ws = (w - beta) / alpha

                step = 2 ** (bit)-1

                # Quantize ws with a linear quantizer to "bit" bits
                R = (1 / step) * torch.round(step * ws)

                # Scale the quantized weight R back with alpha and beta
                wq = alpha * R + beta

            # Lab3 (e), Your code here:
            else:

                max_value = torch.abs(w).max()
                min_value = -1 * max_value
                # calculating alpha
                alpha = max_value - min_value

                # calculating beta
                beta = min_value

                # scale w with alpha and beta so that all elements in ws are between -min_value and max_value
                ws = (w - beta) / alpha

                step = 2 ** (bit)-1

                # Quantize ws with a linear quantizer to "bit" bits
                R = (1 / step) * torch.round(step * ws)

                # Scale the quantized weight R back with alpha and beta
                wq = alpha * R + beta
            # Restore zero elements in wq
            wq = wq*weight_mask

        return wq

    @staticmethod
    def backward(ctx, g):
        return g, None, None
```

Figure 12: Coding implementation for STE Class

b): The below table reports the best test accuracy when we set the `Nbits` to different values. As we can tell, the performance is significantly compromised when we set `Nbits = 3` or `Nbits = 2`. The test accuracy doesn't decrease more than ~2% when $Nbits >= 4$

Nbits	Test Accuracy
Pre-Trained	91.51%
6	91.45%
5	91.12%
4	89.72%
3	76.62%
2	8.99%

c): With finetuning, we're able to increase the test accuracy for all `Nbits`. The finetuning seems to be the most efficient on the lower Nbit values 3 & 2 with significant accuracy gains of ~15% and ~75%(!!)

respectively compared to when no finetuning was applied. It would appear that finetuning is a very efficient strategy in preserving the test accuracy even when the Nbit value is low. The difference between the higher Nbit and lower Nbit test accuracy is less stark compared to when no finetuning was applied.

Nbits	Test Accuracy (With Finetuning)
4	91.37%
3	90.52%
2	85.10%

d): Similar to the previous part, finetuning seems to improve the test accuracy relatively more when the Nbits value is lower. However, pruning seems to limit the gains from finetuning capping the test accuracy improvements at ~25% when `Nbits = 3` and `Nbits = 2`. The test accuracy is very poor when `Nbits = 2` achieving a best test accuracy of 35% even when finetuning is applied. Pruning might be a desirable compression technique if model size is a concern, but it might come at the cost of accuracy especially if the Nbits is small.

Nbits	Test Accuracy (Before Finetuning)	Test Accuracy (After Finetuning)
4	88.48%	90.29%
3	62.89%	88.19%
2	10.00%	35.05%

e): Please see the code implementation for symmetric quantization in Figure 12. From the below table we observe that the test accuracy is worse for all values of `Nbits` when we use symmetric quantization, except when `Nbits = 2`. Generally, the difference between the test accuracy of asymmetric and symmetric quantization increases as we decrease the size of `Nbits`. We don't see a huge difference in performance when `Nbits = 6` or `Nbits = 5`, but when `Nbits = 4` or `Nbits = 3` we see a larger discrepancy of -4-5% in performance.

Nbits	Test Accuracy (Asymmetric)	Test Accuracy (Symmetric)
6	91.45%	91.33%
5	91.12%	90.71%
4	89.72%	85.31%
3	76.62%	71.51%
2	8.99%	10.00%