ECE 661: Homework 4 Pruning and Fixed-point Quantization

Frankie Willard

November 2022

1 True/False Questions (30 pts)

Problem 1.1 (3 pts) Generally speaking, the weight pruning does not intervene the weight quantization, as they are orthogonal techniques in compressing the size of DNN models.

True. According to L12 slide 37, pruning doesn't hurt quantization. This is because the techniques cover different aspects of the modeling process and are thus orthogonal.

Problem 1.2 (3 pts) Pruning removes (zeros) part of the weight parameters from the neural network models. When deploying on general hardware platforms (e.g., GPUs), these zeroed parameters don't need to be computed during inference. Thus, we anticipate that training a pruned model on GPUs is much faster than training its original model.

False. According to L12 slide 9, you need specialized hardware to support pruning. It may not bring much speedup on traditional platforms like GPU, even with specifically designed sparse matrix multiplication algorithm (L12 slide 25). Specialized hardware accelerator should be designed to store and compute non-structured sparse weights (L12 slide 25)

Problem 1.3 (3 pts) In deep compression pipeline, even if we skip the quantization step, the pruned model can still be effectively encoded by the following Huffman coding process as pruning greatly reduces the number of weight variables.

False. While Huffman code is a valid coding scheme with the guaranteed shortest expected length for any symbol distribution (L12 slide 42), quantization is required in order to put the values on a fixed scale such that Huffman coding can be performed. Thus, it is necessary.

Problem 1.4 (3 pts) Directly using SGD to optimize a sparsity-inducing regularizer (i.e. L-1, DeepHoyer etc.) with the training loss will lead to exact

zero values in the weight elements, there's no need to apply additional pruning step after the optimization process.

False. One final pruning step with a small constant threshold (e.g. 1e-4) is needed to reach a sparse model, can reach similar sparse level as the iterative pruning (lecture 13 slide 19).

Problem 1.5 (3 pts) Using soft thresholding operator will lead to better results comparing to using L-1 regularization directly as it solves the "bias" problem of L-1.

False. Soft thresholding reveals the "bias" problem of L1, partially solved by SCAD and MCP etc. (L14 slide 16). Thus, it is only revealing the issue, and SCAD and MCP are what is truly solving

Problem 1.6 (3 pts) Group Lasso can lead to structured sparsity on DNNs, which is more hardwarefriendly. The idea of Group Lasso comes from applying L-1 regularization to the L-2 norm of all of the groups.

True. "For ℓ_2 norm to be 0, all elements within the group have to be 0 simultaneously, leading to structured sparsity" (L13 slides 21-22).

Problem 1.7 (3 pts) Proximal gradient descent introduces an additional proximity term to minimize regularization loss in the proximity of weight parameters. The proximity term allows smoother convergence of the overall objective.

True. The added proximity term will allow smoother convergence of the overall objective (lecture 14 slide 6). This is seen empirically later in the lab.

Problem 1.8 (3 pts) Models equipped with early exits allows some inputs to be computed with only part of the model, thus overcoming the issue of overfitting and overthinking.

True. Early exit models help overcome overfitting and overthinking and identify easy data at early stage (Lecture 14 slide 27-28)

Problem 1.9 (3 pts) When training a quantized model with STE, both the forward-pass and the gradient accumulation will be performed with the full-precision weight kept in the training process.

False. Loss is computed with quantized w' (lecture 15 slide 10)

Problem 1.10 (3 pts) Comparing to quantizing all the layers in a DNN to the same precision, mixedprecision quantization scheme can reach higher accuracy with a similar-sized model.

True. Perform mixed-precision quantization (assign different precisions to different layers) can provide better size/latency-accuracy tradeoff than fixed quantization (Lecture 15- slide 28)

2 Lab 1: Sparse optimization of linear models (30 pts)

(a) (4 pts) Theoretical analysis: with learning rate µ, suppose the weight you have after step k is Wk, derive the symbolic formulation of weight W_{k+1} after step k+1 of full-batch gradient descent with x_i, y_i, i ∈ 1, 2, 3. (Hint: note the loss L we have is defined differently from standard MSE loss.)

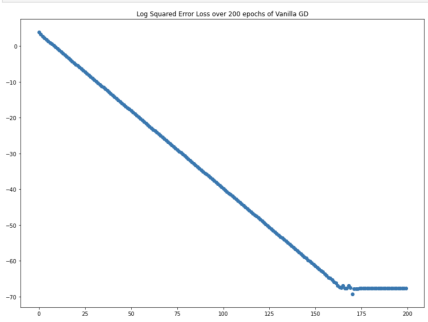
$$W_{k+1} = W_k - \mu \frac{\partial L}{\partial W_k}$$

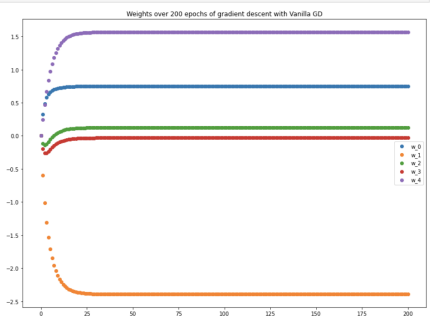
$$= W_k - \mu(2 * X^T(y - \hat{y}))$$

$$= W_k - \mu\left(2 * \begin{bmatrix} x_1^T & x_2^T & x_3^T \end{bmatrix} \left(\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} - \begin{bmatrix} \hat{y_1} \\ \hat{y_2} \\ \hat{y_3} \end{bmatrix} \right)\right)$$

(b) (3 pts) In Python, directly minimize the objective L without any sparsity-inducing regularization/constraint. Plot the value of log(L) vs. steps throughout the training, and use another figure to plot how the value of each element in W is changing throughout the training. From your result, is W converging to an optimal solution? Is W converging to a sparse solution?

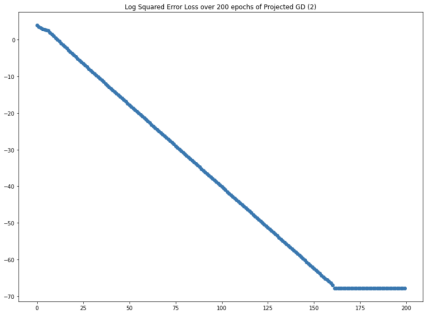
From my result, W is converging to an optimal solution, however, it is not sparse. The loss continually reduces to a miniscule value and then plateaus at the optimal solution. However, the weights themselves are not zero, but rather: array([0.74759615, -2.38942308, 0.12259615, -0.03125 , 1.56490385]). Of these weights, none of them are zero (and only one is close to zero), such that this would not be considered a sparse solution.

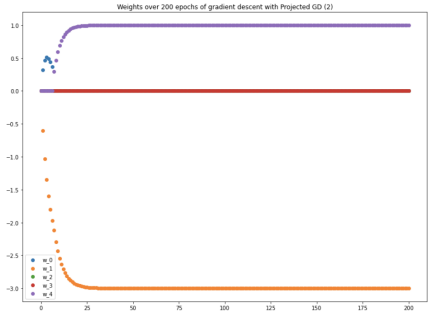




(c) (6 pts) Since we have the knowledge that the ground-truth weight should have $||W||_0 \leq 2$, we can apply projected gradient descent to enforce this sparse constraint. Redo the optimization process in (b), this time prune the elements in W after every gradient descent step to ensure $||W^l||_0 \leq 2$. Plot the value of log(L) throughout the training, and use another figure to plot the value of each element in W in each step. From your result, is W converging to an optimal solution? Is W converging to a sparse solution?

From my result, W is converging to an optimal solution, that is sparse. The loss continually reduces to a miniscule value (very similar to the regular GD) and then plateaus at the optimal solution. Additionally, three of the five weights are zero, producing array([0., -3., -0., -0., 1.]). Thus, this would be considered a sparse solution.

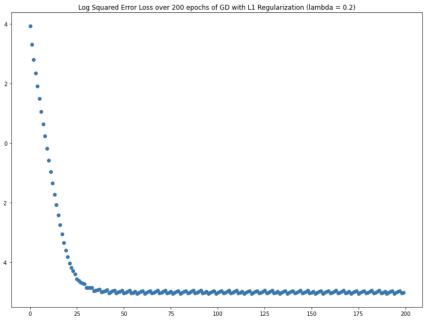


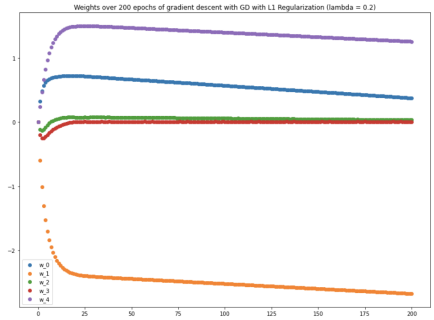


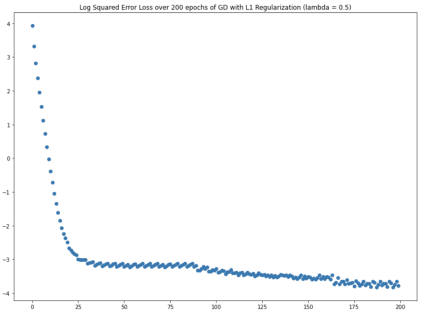
(d) (5 pts) In this problem we apply ℓ_1 regularization to induce the sparse solution. The minimization objective therefore changes to L + λ $||W||_1$. Please use full-batch gradient descent to minimize this objective, with λ = 0.2, 0.5, 1.0, 2.0 respectively. For each case, plot the value of log(L) throughout the training, and use another figure to plot the value of each element in W in each step. From your result, comment on the convergence performance under different λ.

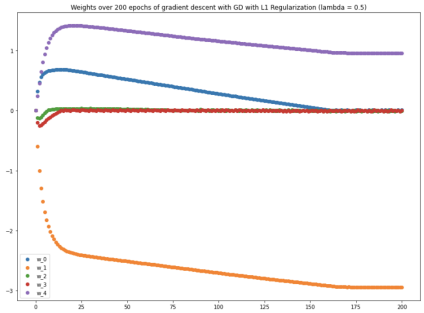
As we increase lambda, we find that we increase the bumpiness/instability in the convergence performance, and continuously perform worse in the final output for each lambda. This is because we are plotting the squared error objective, and as we increase the lambda, we increase the emphasis on the L1 norm added to the objective function being optimized. This leads to the

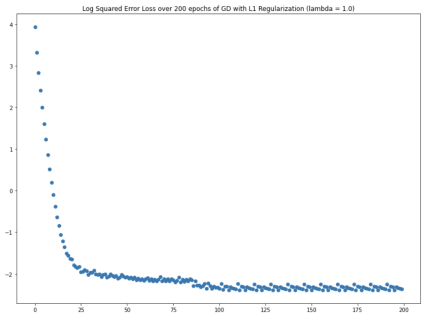
models with higher lambdas to have more coefficients closer to 0 (to reduce the L1 norm). In doing so, this sometimes leads to slight increases in the squared error objective (as we are slightly increasing bias to reduce variance by introducing a sparser solution). The instability increasing comes from the fact that these models are working to optimize two different contrasting metrics, such that the more it focuses on one, the more it neglects the other (leading to decreases in weights increasing bias leading to a gradient step that decreases the squared error loss and so on as the loss gets smaller and smaller). The final loss value is reached earlier for higher lambdas (as seen by the sharp decrease followed by a flat trend of bumpiness)

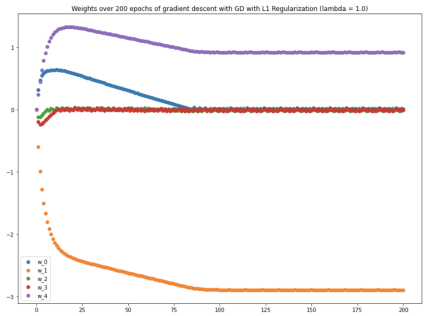


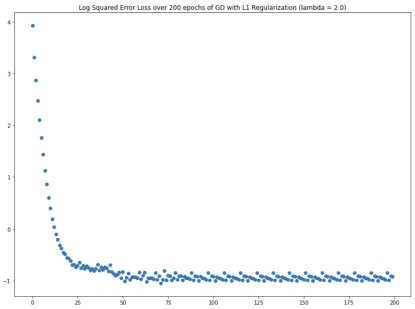


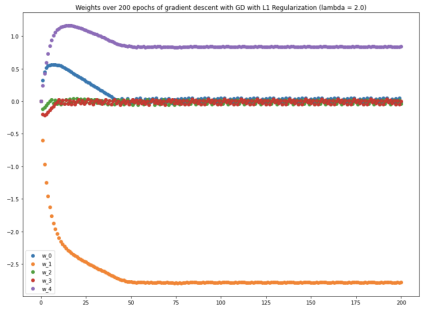








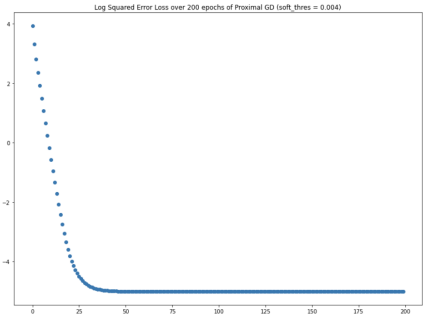


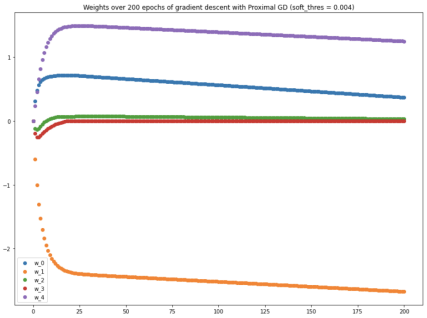


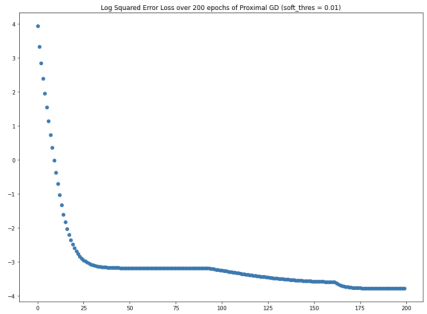
(e) (6 pts) Here we optimize the same objective as in (d), this time using proximal gradient update. Recall that the proximal operator of the ℓ_1 regularizer is the soft thresholding function. Set the threshold in the soft thresholding function to 0.004, 0.01, 0.02, 0.04 respectively. Plot the value of log(L) throughout the training, and use another figure to plot the value of each element in W in each step. Compare the convergence performance with the results in (d). (Hint: Optimizing L + λ $||W||_1$ using gradient descent with learning rate μ should correspond to proximal gradient update with threshold $\mu\lambda$)

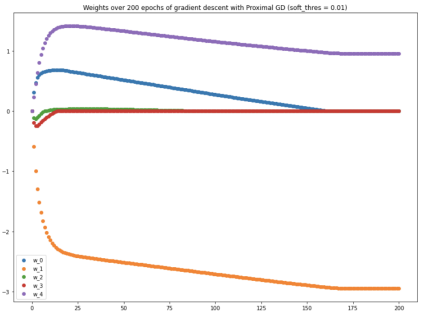
As we increase the soft thresholding function being passed in, we find similar results to that of d, in terms of the higher final losses at the end (and similar

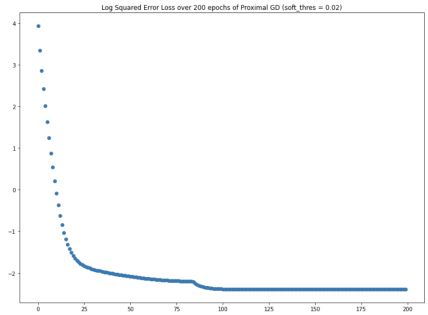
loss function shapes). However, the final convergence is a much smoother path that does not bounce back and forth as much as in d. This is an intentional byproduct of proximal gradient updates, which are meant to allow smoother convergence of the overall objective (L14 slide 6)

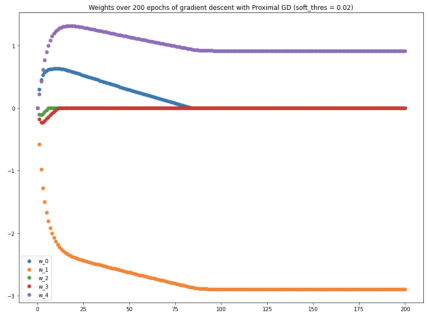


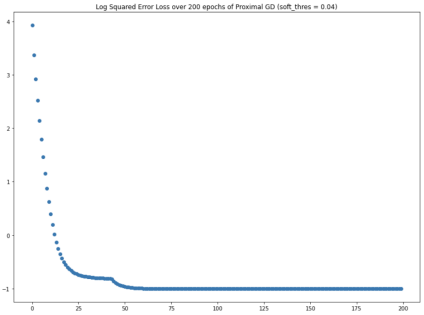


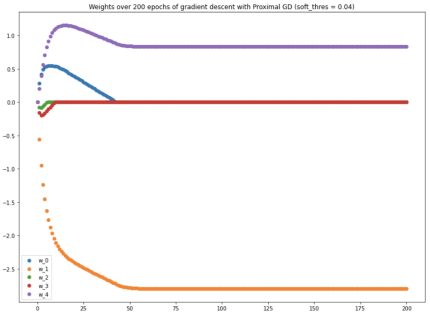






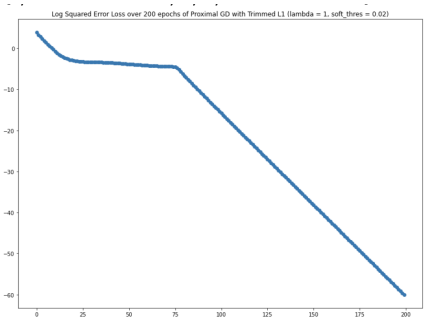


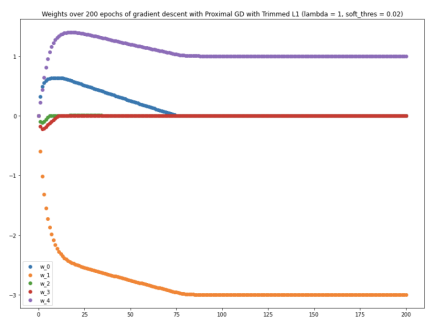


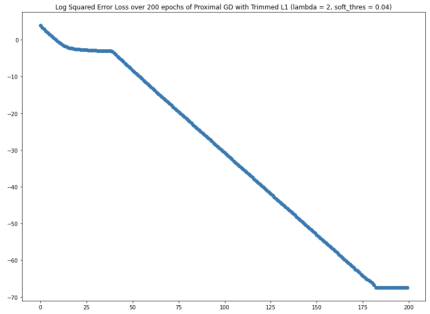


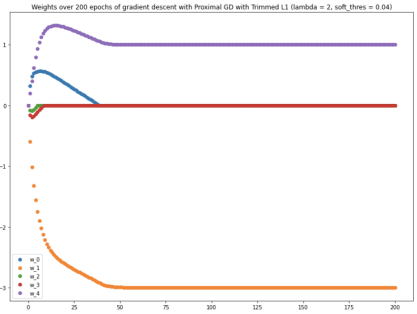
(f) (6 pts) Trimmed ℓ_1 (T ℓ_1) regularizer is proposed to solve the "bias" problem of ℓ_1. For simplicity you may implement the T ℓ_1 regularizer as applying a ℓ_1 regularization with strength λ on the 3 elements of W with the smallest absolute value, with no penalty on other elements. Minimize L+λT ℓ_1(W) using proximal gradient update with $\lambda = 1.0, 2.0, 5.0, 10.0$ (correspond the soft thresholding threshold 0.02, 0.04, 0.1, 0.2). Plot the value of log(L) throughout the training, and use another figure to plot the value of each element in W in each step. Comment on the convergence comparison of the Trimmed ℓ_1 and the ℓ_1. Also compare the behavior of the early steps (e.g. first 20) between the Trimmed ℓ_1 and the iterative pruning.

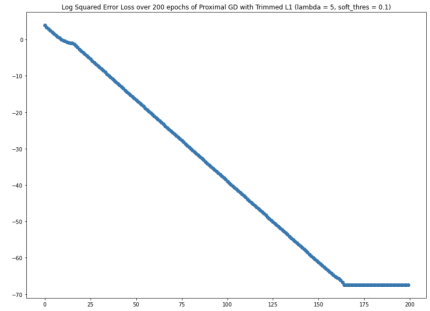
In terms of the convergence, I find that the trimmed L1 norm does converges better than the L1 norm in terms of the final loss value, reaching much lower values (on the order of 10 to the -60 or -70). The early steps for the trimmed (for the smaller parameters) have a much slower decrease (curves off to the right rather than straight down) in loss at the beginning when compared to all other techniques. This is then followed by a plateau in loss for awhile, before a sharp decrease experienced later.

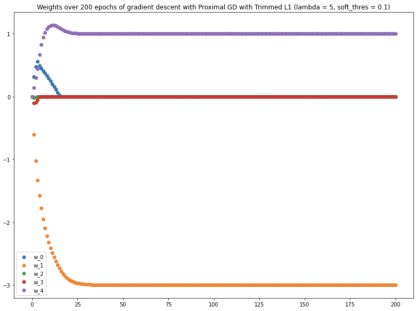

Log Squared Error Loss over 200 epochs of Proximal GD with Trimmed L1 (lambda = 1, soft_thres = 0.02)

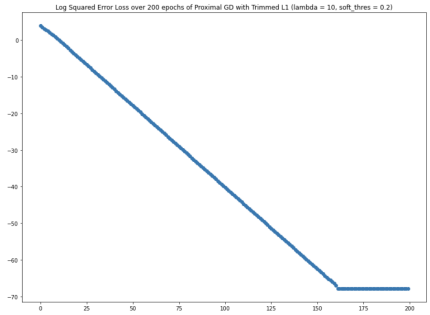

Weights over 200 epochs of gradient descent with Proximal GD with Trimmed L1 (lambda = 1, soft_thres = 0.02)
w_0
w_1
w_2
w_3
w_4

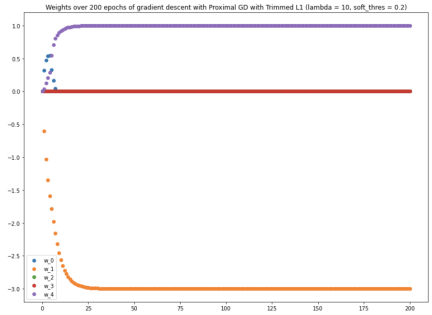

Log Squared Error Loss over 200 epochs of Proximal GD with Trimmed L1 (lambda = 2, soft_thres = 0.04)


Weights over 200 epochs of gradient descent with Proximal GD with Trimmed L1 (lambda = 2, soft_thres = 0.04)
w_0
w_1
w_2
w_3
w_4


Log Squared Error Loss over 200 epochs of Proximal GD with Trimmed L1 (lambda = 5, soft_thres = 0.1)


Weights over 200 epochs of gradient descent with Proximal GD with Trimmed L1 (lambda = 5, soft_thres = 0.1)
w_0
w_1
w_2
w_3
w_4


Log Squared Error Loss over 200 epochs of Proximal GD with Trimmed L1 (lambda = 10, soft_thres = 0.2)


Weights over 200 epochs of gradient descent with Proximal GD with Trimmed L1 (lambda = 10, soft_thres = 0.2)
w_0
w_1
w_2
w_3
w_4

3 Lab 2: Pruning ResNet-20 model (30 pts)

(a) (2 pts) In hw4.ipynb, run through the first three code block, report the accuracy of the floatingpoint pretrained model.

Test accuracy=0.9151

(b) (8 pts) Complete the implementation of pruning by percentage function in the notebook. Here we determines the pruning threshold in each DNN layer by the 'q-th percentile' value in the absolute value of layer's weight element. Use the next block to call your implemented pruning by percentage. Try pruning percentage q = 0.4, 0.6, 0.8. Report the test accuracy q. (Hint: You need to reload the full model checkpoint before applying the prune function with a different q).

q	Test Accuracy
0.4	0.8874
0.6	0.7223
0.8	0.1003

(c) (6 pts) Fill in the $finetune\_after\_prune$ function for pruned model finetuning. Make sure the pruned away elements in previous step are kept as 0 throughout the finetuning process. Finetune the pruned model with q=0.8 for 20 epochs with the provided training pipeline. Report the best accuracy achieved during finetuning. Finish the code for sparsity evaluation to check if the finetuned model preserves the sparsity

Best accuracy=0.8799

The sparsity is preserved in the finetuned model as demonstrated below

(d) (6 pts) Implement iterative pruning. Instead of applying single step pruning before finetuning, try iteratively increase the sparsity of the model before each epoch of finetuning. Linearly increase the pruning percentage for 10 epochs until reaching 80 percent in the final epoch (prune $(8 \times e)$ percent before epoch e) then continue finetune for 10 epochs. Pruned weight can be recovered during the iterative pruning process before the final pruning step. Compare performance with (c)

Best accuracy=0.8756

Based on the accuracies, we find that iterative pruning performs marginally worse than the answer in (c), which is about 0.0043.

(e) (8 pts) Perform magnitude-based global iterative pruning. Previously we set the pruning threshold of each layer following the weight distribution of

```
Sparsity of head_conv.0.conv: 0.7986111111111112
Sparsity of body_op.0.conv1.0.conv: 0.7999131944444444
Sparsity of body_op.0.conv2.0.conv: 0.7999131944444444
Sparsity of body_op.1.conv1.0.conv: 0.7999131944444444
Sparsity of body_op.1.conv2.0.conv: 0.7999131944444444
Sparsity of body_op.2.conv1.0.conv: 0.7999131944444444
Sparsity of body_op.2.conv2.0.conv: 0.7999131944444444
Sparsity of body_op.3.conv1.0.conv: 0.7999131944444444
Sparsity of body_op.3.conv2.0.conv: 0.7999131944444444
Sparsity of body_op.4.conv1.0.conv: 0.7999131944444444
Sparsity of body_op.4.conv2.0.conv: 0.7999131944444444
Sparsity of body_op.5.conv1.0.conv: 0.7999131944444444
Sparsity of body_op.5.conv2.0.conv: 0.7999131944444444
Sparsity of body_op.6.conv1.0.conv: 0.7999674479166666
Sparsity of body_op.6.conv2.0.conv: 0.7999945746527778
Sparsity of body_op.7.conv1.0.conv: 0.7999945746527778
Sparsity of body_op.7.conv2.0.conv: 0.7999945746527778
Sparsity of body_op.8.conv1.0.conv: 0.7999945746527778
Sparsity of body_op.8.conv2.0.conv: 0.7999945746527778
Sparsity of final_fc.linear: 0.8
Files already downloaded and verified
Test Loss=0.3648, Test accuracy=0.8799
```

the layer, and prune all layers to the same sparsity. This will constraint the flexibility in the final sparsity pattern across layers. In this question, Fill in the *global_prune_by_percentage* function to perform a global ranking of the weight magnitude from all the layers, and determine a single pruning threshold by percentage for all the layers. Repeat iterative pruning to 80 percent sparsity, report final accuracy and the percentage of zeros in each layer.

Best accuracy=0.8840

4 Lab 3: Fixed-point quantization and finetuning (20+10 pts)

(a) (10 pts) As is mentioned in lecture 15, to train a quantized model we need to use floatingpoint weight as trainable variable while use a straight-through estimator (STE) in forward and backward pass to convert the weight into quantized value. Intuitively, the forward pass of STE converts a float weight into fixed-point, while the backward pass passes the gradient straightly through the quantizer to the float weight.

To start with, implement the STE forward function in *FP_layers.py*, so that it serves as a linear quantizer with dynamic scaling, as introduced on

```
Sparsity of head_conv.0.conv: 0.3101851851851852
Sparsity of body_op.0.conv1.0.conv: 0.6566840277777778
Sparsity of body_op.0.conv2.0.conv: 0.6384548611111112
Sparsity of body_op.1.conv1.0.conv: 0.6258680555555556
Sparsity of body_op.1.conv2.0.conv: 0.6480034722222222
Sparsity of body_op.2.conv1.0.conv: 0.6310763888888888
Sparsity of body_op.2.conv2.0.conv: 0.6692708333333334
Sparsity of body_op.3.conv1.0.conv: 0.6243489583333334
Sparsity of body_op.3.conv2.0.conv: 0.6886935763888888
Sparsity of body_op.4.conv1.0.conv: 0.7253689236111112
Sparsity of body_op.4.conv2.0.conv: 0.7822265625
Sparsity of body_op.5.conv1.0.conv: 0.7248263888888888
Sparsity of body_op.5.conv2.0.conv: 0.8130425347222222
Sparsity of body_op.6.conv1.0.conv: 0.7327473958333334
Sparsity of body_op.6.conv2.0.conv: 0.7643500434027778
Sparsity of body_op.7.conv1.0.conv: 0.7770182291666666
Sparsity of body_op.7.conv2.0.conv: 0.8261176215277778
Sparsity of body_op.8.conv1.0.conv: 0.8528103298611112
Sparsity of body_op.8.conv2.0.conv: 0.9767795138888888
Sparsity of final_fc.linear: 0.1578125
Total sparsity of: 0.7999970186631685
Files already downloaded and verified
Test Loss=0.3483, Test accuracy=0.8840
```

page 9 of lecture 15. Please follow the comments in the code to figure out the expected functionality of each line. Take a screen shot of the finished STE class and paste it into the report. Submission of the *FP_layers.py* file is not required.

```python
class STE(torch.autograd.Function):
    @staticmethod
    def forward(ctx, w, bit):
        if bit is None:
            wq = w
        elif bit==0:
            wq = w*0
        else:
            # For Lab 3 bouns only (optional), build a mask to record position of zero weights
            weight_mask = np.where(w == 0, 1, 0)

            # Lab3 (a), Your code here:
            # Compute alpha (scale) for dynamic scaling
            alpha = torch.max(w) - torch.min(w)
            # Compute beta (bias) for dynamic scaling
            beta = torch.min(w)
            # 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

            # For Lab 3 bouns only (optional), restore zero elements in wq
            wq = wq*weight_mask

        return wq
```

(b) (4 pts) In hw4.ipynb, load pretrained ResNet-20 model, report the accuracy of the floating-point pretrained model. Then set Nbits in the first line of block 4 to 6, 5, 4, 3, and 2 respectively, run it and report the test accuracy you got. (Hint: In this block the line defining the ResNet model (second line) will set the residual blocks in all three stages to Nbits fixed-point, while keeping the first conv and final FC layer still as floating point.)

Pre-trained accuracy=0.9151

Nbits	Test Accuracy
6	0.9145
5	0.9112
4	0.8972
3	0.7662
2	0.0899

(c) (6 pts) With Nbits set to 4, 3, and 2 respectively, run code block 4 and 5 to finetune the quantized model for 20 epochs. You do not need to change other parameter in the finetune function. For each precision, report the highest testing accuracy you get during finetuning. Comment on the relationship between precision and accuracy, and on the effectiveness of finetuning.

The higher the Nbits value, the higher accuracy we find in both problems b and c. There is a small decrease in accuracy from 6 bits to 5 bits, but this de-

crease increases with each bit lost (this is intuitive as you are losing a higher percentage of information relative to what you had as 6/5 ¡ 5/4). There is a key precision-accuracy tradeoff that suggests lower precision (which is computationally nice with storage) comes with lower accuracy. We find that fine tuning worked quite well, as the 4 bit case increased marginally (to be close to the same as 6 bits without tuning), the 3 bit case increased from 0.7662 to 0.9074 (a large performance increase making it similar to 4-5 bits without tuning), and the 2 bit case made an incredibly large increase to from 0.0899 to 0.8600 (just under the performance of 4 bits without tuning). Thus, the tuning is more helpful for the lower amount of bits, and is able to get you the same performance as with 1-2 more bits and no tuning.

Nbits	Test Accuracy
4	0.9135
3	0.9074
2	0.8600

(d) (Bonus 5 pts) In practice, we want to apply both pruning and quantization on the DNN model. Here we explore how pruning will affect quantization performance. Please load the checkpoint of the 80 percent sparsity model with the best accuracy from Lab 2, repeat the process in (c), report the accuracy before and after finetuning, and discuss your observations comparing to (c)'s results. (Hint: Please consider zeros in the weight as being pruned away, and only apply STE on nonzero weight elements for quantization. Modification on STE implementation may be needed. For analysis you may focus on the accuracy drop after quantization.)

Using this technique, we found worse performance for each Nbits amount than in c. For the 4 bit and 3 bit case, this is a small decrease of about 0.02-0.3 in performance which may not be substantial for a given problem (given the benefit of compression in return). However, in the 2 bit case, this decrease is about 0.56, which is a massive performance drop. The finetuning worked very well for these models, drastically increasing their performance.

Nbits	Before Test Accuracy	After Test Accuracy
4	0.1000	0.8974
3	0.1015	0.8752
2	0.1090	0.3092

(e) (Bonus 5 pts) Another way to have both pruning and quantization is to directly apply quantization aware training during the iterative pruning process. Starting from the pretrained ResNet-20 model, repeat the iterative pruning process of Lab 2(e) with 4-bit quantization. Report the final accuracy and the percentage of zeros in each layer. Compare your result with Lab2(e) results and the result of the previous question and discuss your observations.

Using this technique, we found much worse performance than Lab 2e. While Lab 2e got an accuracy of 0.884, this tehcnique got an accuracy of 0.4125,

which is a massive performance drop. In comparing to last question, we used 4 bit quantization here, which in the last question, was able to garner a performance of about 0.8974. Thus, this technique is far worse in terms of maintaining performance while achieving the goal of quantization.

Test accuracy=0.4125

```
Sparsity of head_conv.0.conv: 0.30787037037037035
Sparsity of body_op.0.conv1.0.conv: 0.6575520833333334
Sparsity of body_op.0.conv2.0.conv: 0.6397569444444444
Sparsity of body_op.1.conv1.0.conv: 0.6280381944444444
Sparsity of body_op.1.conv2.0.conv: 0.6488715277777778
Sparsity of body_op.2.conv1.0.conv: 0.6315104166666666
Sparsity of body_op.2.conv2.0.conv: 0.6684027777777778
Sparsity of body_op.3.conv1.0.conv: 0.6252170138888888
Sparsity of body_op.3.conv2.0.conv: 0.6882595486111112
Sparsity of body_op.4.conv1.0.conv: 0.7261284722222222
Sparsity of body_op.4.conv2.0.conv: 0.7820095486111112
Sparsity of body_op.5.conv1.0.conv: 0.7249348958333334
Sparsity of body_op.5.conv2.0.conv: 0.8132595486111112
Sparsity of body_op.6.conv1.0.conv: 0.7331814236111112
Sparsity of body_op.6.conv2.0.conv: 0.7640245225694444
Sparsity of body_op.7.conv1.0.conv: 0.7770453559027778
Sparsity of body_op.7.conv2.0.conv: 0.8261176215277778
Sparsity of body_op.8.conv1.0.conv: 0.8527018229166666
Sparsity of body_op.8.conv2.0.conv: 0.9764539930555556
Sparsity of final_fc.linear: 0.159375
Total sparsity of: 0.7999970186631685
Files already downloaded and verified
Test Loss=2.2337, Test accuracy=0.4125
```