Anthony DiSpirito

ECE 590 – Comp. Eng. M.L. and D.N.Ns

Dr. Yiran Chen

11/8/19

Lab 3:

Pruning, Quantization, and Huffman Coding to Compress DNNs

**NOTE:** Code and Documentation can also be found at github.com under my Github account name “axd465” and my repo “ECE590-CompEngML-DL-Lab3”

**Link:** [**https://github.com/axd465/ECE590-CompEngML-DL-Lab3**](https://github.com/axd465/ECE590-CompEngML-DL-Lab3)

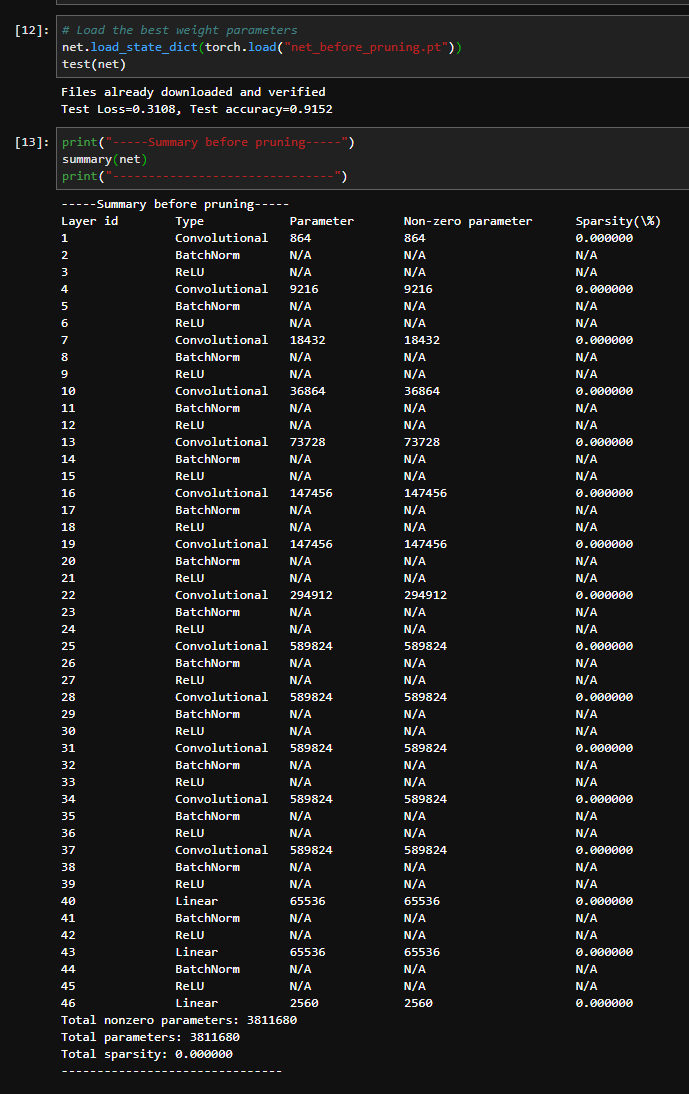
Assignment 1:

* For this problem, my goal was to achieve the best model possible, to give myself the best starting chance for the rest of the project. The only parameters we were able to modify were: learning rate, epochs, and the L2 regularization (which promotes many non-zero smaller magnitude weights). Note here that batch size was always kept consistent at 256. From previous experience, I knew that the most important factor for tuning would be the learning rate. I took a look at the train\_util.py document to see the optimization algorithm was SGD with momentum (not Nesterov). From the slides and previous knowledge, I knew a good starting point would be the default value of initial learning rate = 0.01 and reg = 5e-4. From previous experience I knew that I should start with epochs = 20, to establish general trends. What followed were a serious of experiments and results recorded in the first table below, where the regularization was kept constant and the learning rate was modified. I chose what seemed like the best learning rate (between 0.06 and 0.05 at 0.055) and kept that constant when modifying the regularization.

|  |  |  |  |
| --- | --- | --- | --- |
| TABLE 1: REG CONSTANT | | | |
| LR | REG | Final Test Acc | Final Test Loss |
| 0.01 | 5e-4 | 0.8484 | 0.4450 |
| 0.05 | 5e-4 | 0.8656 | 0.3147 |
| 0.005 | 5e-4 | 0.8237 | 0.51 |
| 0.1 | 5e-4 | 0.8596 | 0.4204 |
| 0.075 | 5e-4 | 0.8640 | 0.4135 |
| 0.06 | 5e-4 | 0.8650 | 0.4032 |

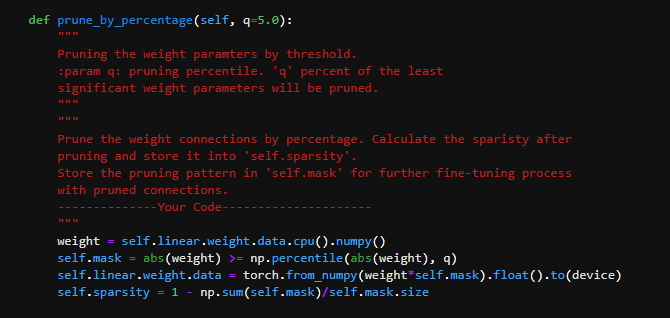
|  |  |  |  |
| --- | --- | --- | --- |
| TABLE 2: LR CONSTANT | | | |
| LR | REG | Final Test Acc | Final Test Loss |
| 0.055 | 3e-4 | 0.8626 | 0.4171 |
| 0.055 | 10e-4 | 0.8717 | 0.3911 |
| 0.055 | 8e-4 | 0.8718 | 0.3908 |

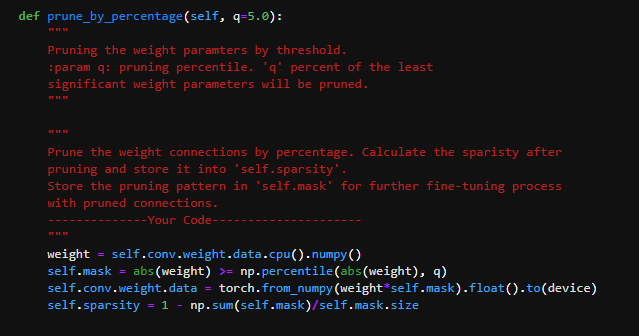
* From these tests, I decided to try and reach above 90% with LR = 0.055 and REG = 8e-4 by training for 75 epochs as this combination seemed to converge smoothly without bottoming out too early. This resulted in a final testing accuracy of 0.9152 (as depicted below, with model summary). As this was well above 90%, I decided my tuning was done and moved on.



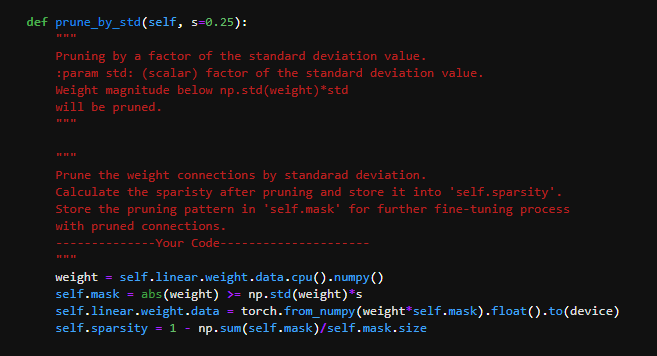
Assignment 2:

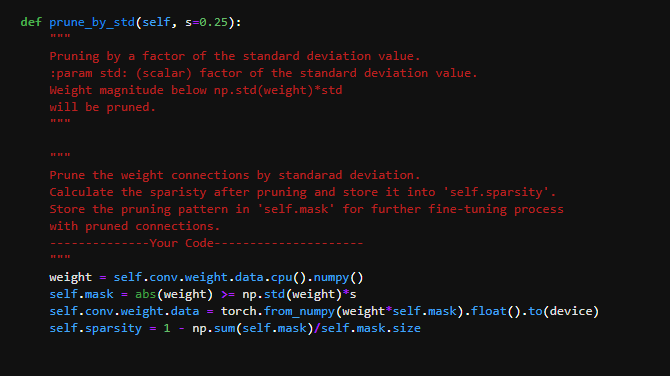
1. See below code for the completed prune\_by\_percentage sections of the pruned\_layers.py document (in both PruneLinear and PruneConv respectively).



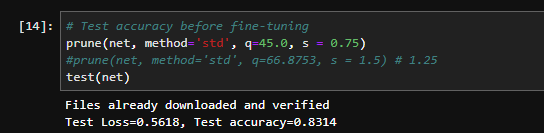


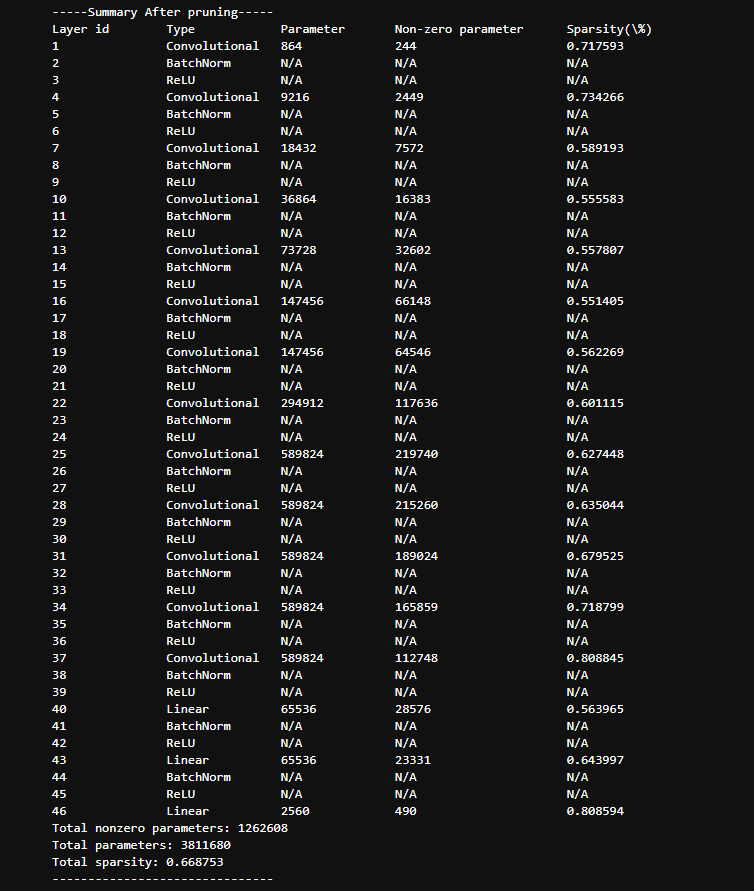
1. See below code for completed prune\_by\_std sections of the pruned\_layers.py document (in both PruneLinear and PruneConv respectively).



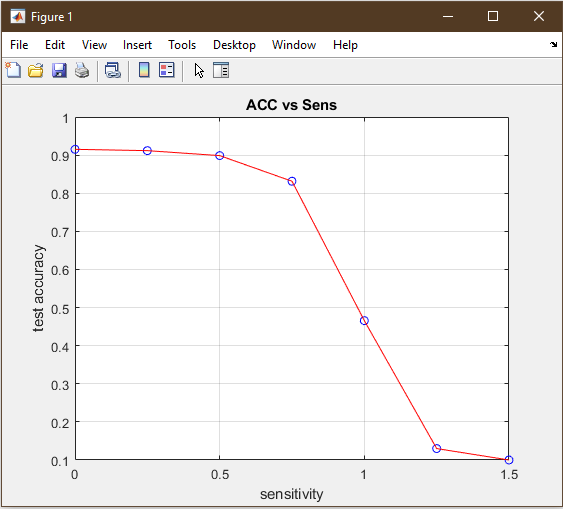


1. After pruning with the std method and s = 0.75, my test accuracy was 0.8314 (as shown below), which is down from my previous result of 0.9152. Below is also depicted the model summary, with a total sparsity of 0.668753. This drop in accuracy is likely due to the fact that some weights (67%) that previously held significance within the model have now been zeroed without retraining. The only reason the accuracy did not drop lower is because on average the larger magnitude weights (unpruned) are more important.

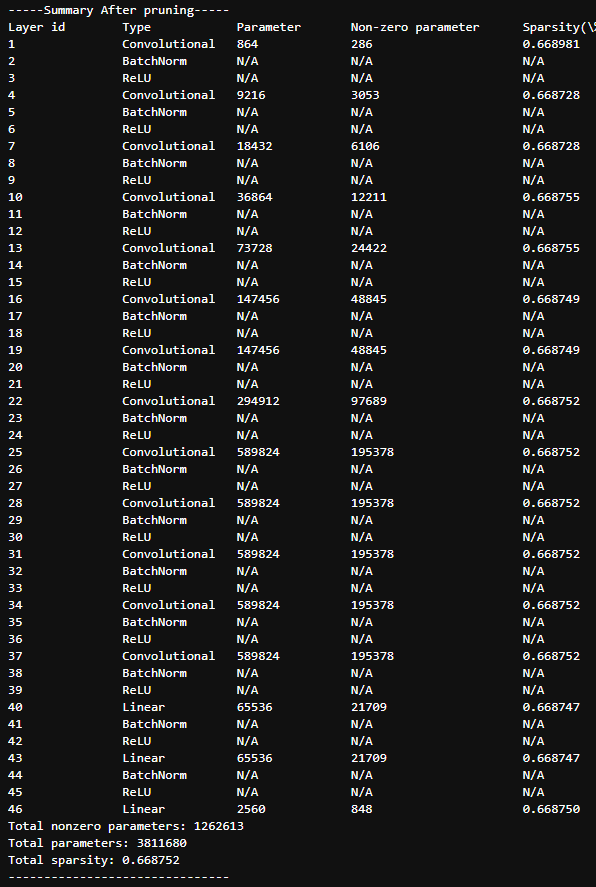




1. Upon testing multiple values for the sensitivity I found the below relationship to test accuracy. At first, the change in sensitivity has little effect, then there is a critical point of dramatic accuracy drop, followed by the bottoming out of accuracy at the near random point. It seems there is a tolerable range of sparsity before the model needs to be retrained/fine-tuned (about 0.75).

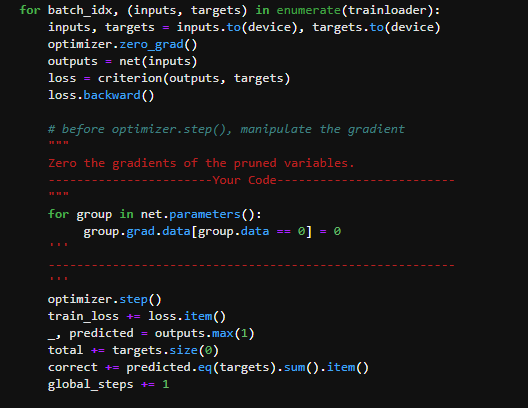


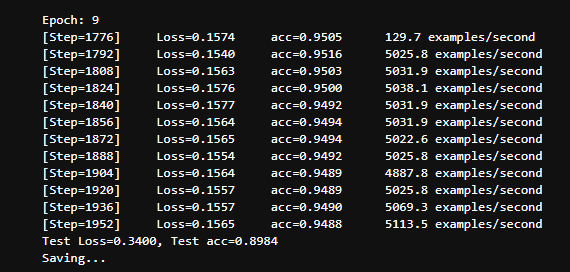
1. As seen below, after changing my method to percentage, I was able to achieve a comparable sparsity to the std method at q = 66.8753 (just inputted the final sparsity desired so they were equal). However, as seen below, the test accuracy was just 0.6816 as compared to the std method which had a test accuracy of 0.8314 with the same total sparsity. This makes sense, as the std method is able to vary by layer and is tied to the variance of the normal weight distribution, as opposed to having a hard threshold imposed on all the weight distributions (see below where every layer had nearly the same sparsity). This lack of variability layer by layer, distribution by distribution caused more of the important weights to be pruned away.



Assignment 3:

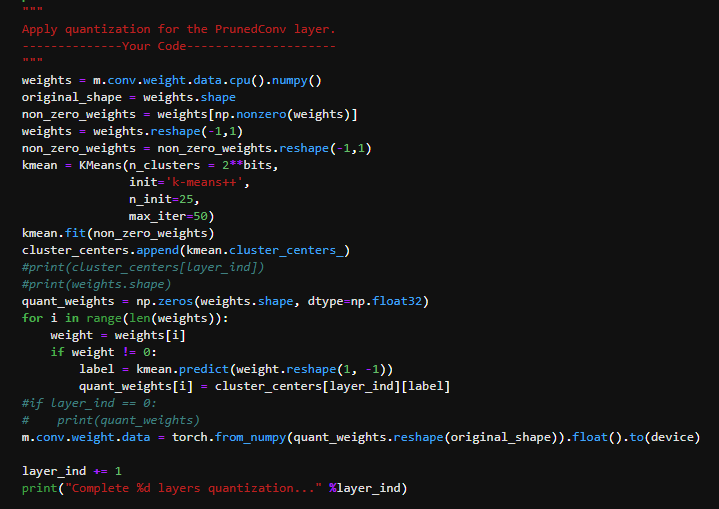
1. See below as well as the attached code how I completed the train\_util.py document to zero out the pruned gradients (so they remain at zero). After fine-tuning/retraining with the stock hyperparameters (which seemed reasonable – smaller initial learning rate and smaller L2 regularization) and s = 1.25 for the std method (total sparsity of 0.8424) for 10 epochs, I got a test accuracy of 0.8984. This was well above the 0.13 I was getting without fine-tuning. I would say this method could probably recover most of the accuracy lost in pruning.

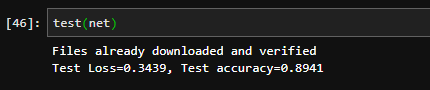




Assignment 4:

1. Below is how I performed the quantization procedure in the code (PrunedConv and linear are nearly identical). After quantizing to 5 bits via weight sharing on my aforementioned 0.84 sparsity model, my test accuracy was 0.8941 (as seen below). The quantization of the sparse weights seems to have very little impact on the test accuracy of the model when quantizing to 5 bits. This is likely because the pruned weight distribution already has some decreased variability (kind of like the bottleneck effect in biological populations). Therefore, it should be relatively straightforward to group these weights to similar values when using 25 = 32 clusters. This is almost like a sampling/discretization procedure on the weight distribution, with the bottleneck effect probably being the best analogy.



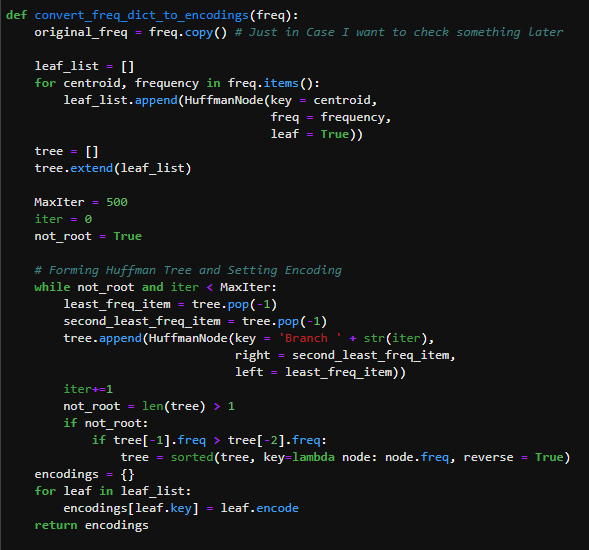


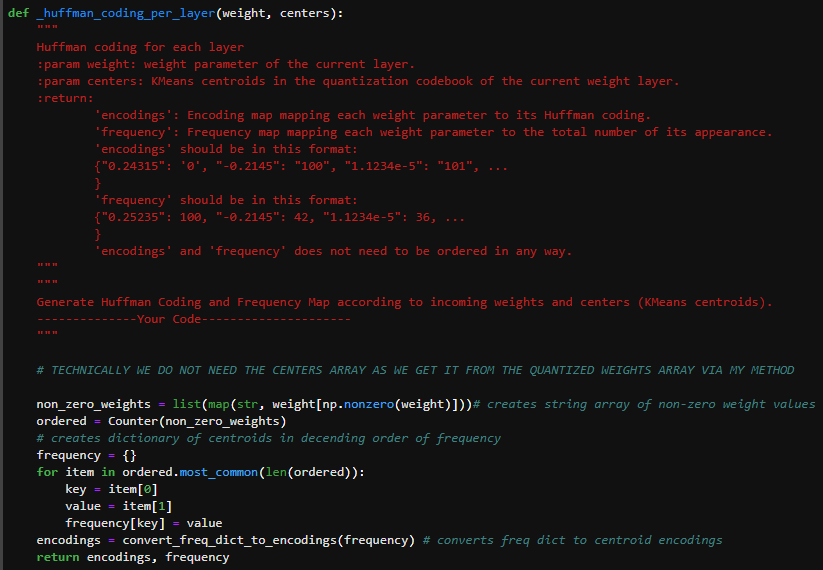
1. After playing around with the number of quantization bits, I found the following results for different quantization amounts [bits = 2, acc = 0.7308], [bits = 3, acc = 0.8675], and [bits = 4, acc = 0.8912]. The optimal bit here seems to be bits = 3, which is about 3% less accuracy than the 5 bit quantization with a quarter of the bits. At 2 bits there is a steep drop-off, so 3 bits seems optimal (especially because this is without retraining/ quantization).

Assignment 5:

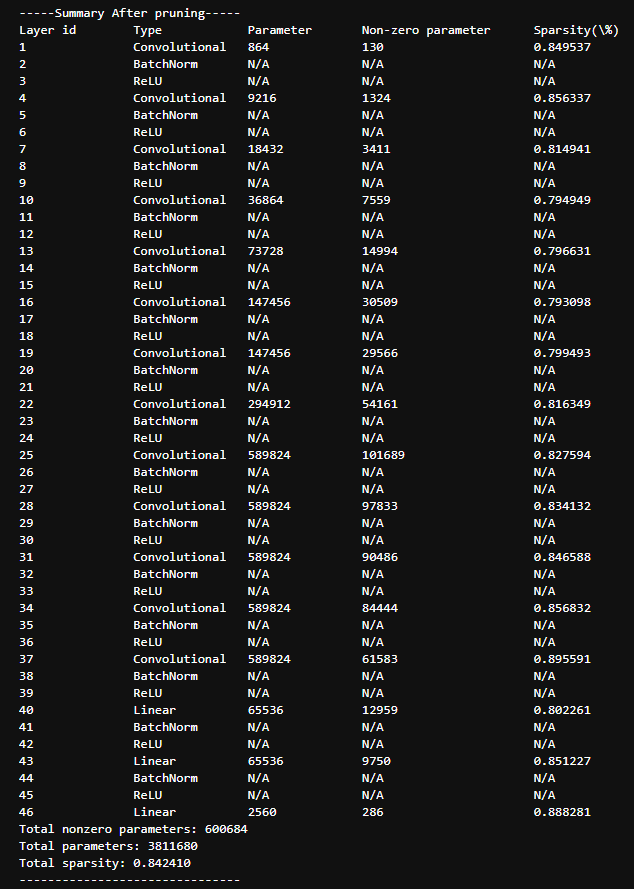
1. The Huffman coding reduces the memory footprints of DNNs because it can compress the number of bits necessary to store the values in the weight matrix. It does this by encoding the weights based on the frequency of occurrence. Those weights with higher frequency get shorter encodings. For example, let’s say you have a matrix filled entirely with a two or three large numbers. Every time those numbers occur as weights, you would normally need to store that large number again. This can get very expensive. Using Huffman encoding, you could create a Huffman table (relatively low cost for this example) and simply store these weights as a much smaller footprint encoding. Thus reducing the total memory consumption dramatically.
2. Below is how I implemented Huffman coding via a Huffman node object and the creation of a tree. I also thought of a clever idea that appended to the leaf encodings every time overarching branches merged. My Huffman object stores each node in the Huffman tree. The leaves have an attribute (leaf=True), that is used when appending an addition to the encoding. Each node has a left and right branch, where these values are set to None for the leaves. I then use the Huffman coding algorithm to remove leaves from a queue (those with the least two frequencies), and form a new Huffman node out of that. The differentiation between leaves and other such nodes is made explicit in the instantiation. Non-leaves have pointers to other Huffman objects assigned to left and right and their frequencies are determined by adding the left and right frequencies. Whenever a node combination occurs, I recursively go down the right and left node pointers to add the appropriate encoding addition to the leaves. After node combination, the top node object is placed back into the queue (which is then resorted by frequency). After the queue only has the root left, I access the encodings via a list of pointers to the original leaf objects.

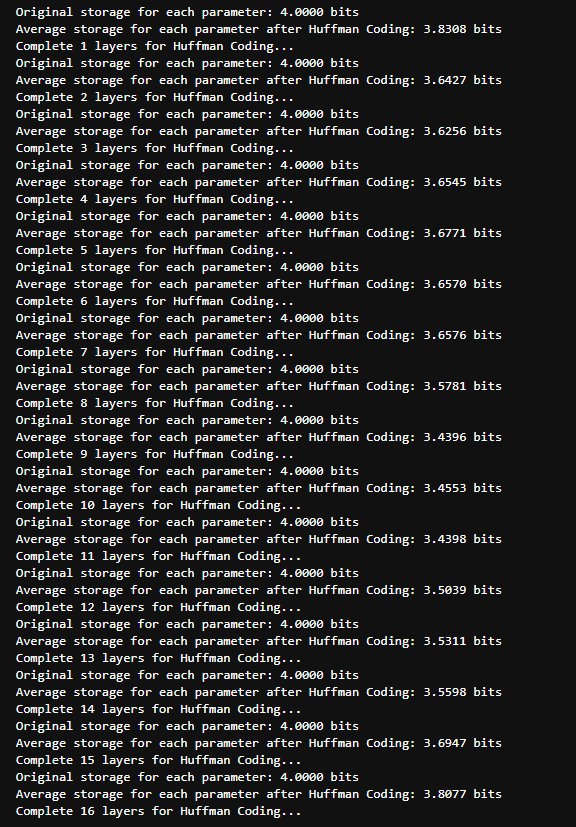






1. Below is the quantitative analysis of the additional memory reduction with the usage of Huffman coding and the calculated average encoding length (per layer as a weighted average). The calculation below uses the 84% sparsity, 4 bit quantized model.
   * Layer 1: (3.8308 bits per param \* 130 param) = 498.004 bits
   * Layer 2: (3.6427 \* 1324) = 4822.9348 bits
   * Layer 3: (3.6256 \* 3411) = 12366.9 bits
   * Layer 4: (3.6545 \* 7559) = 27624.4 bits
   * Layer 5: (3.6771 \* 14994) = 55134.4 bits
   * Layer 6: (3.6570 \* 30509) = 111571.4 bits
   * Layer 7: (3.6576 \* 29566) = 108140.6 bits
   * Layer 8: (3.5781 \* 54161) = 193793.5 bits
   * Layer 9: (3.4396 \* 101689) = 349769 bits
   * Layer 10: (3.4553 \* 97833) = 338042 bits
   * Layer 11: (3.4398 \* 90486) = 311253 bits
   * Layer 12: (3.5039 \* 84444) = 295883.3 bits
   * Layer 13: (3.5311 \* 61583) = 217455.7 bits
   * Layer 14: (3.5598 \* 12959) = 46131.4 bits
   * Layer 15: (3.6947 \* 9750) = 36032.3 bits
   * Layer 16: (3.8077 \* 286) = 1089 bits
   * Average Encoding Length = sum(Layer Lengths)/total\_num\_param = 2103607.84/600684 = 3.502 bits
   * Memory Reduction (%): = 1 – average\_Huffman\_length/original\_length x 100% = 1 – 3.502/4 x 100% = 12.45% reduction in memory





Assignment 6:

* See Below for depiction of accuracy at various stages, as well as parameters used and final accuracy after compression (final accuracy reducing compression step is the quantization).
* In order to achieve this final accuracy and compression ratio, I implemented a couple tricks in addition to those already discussed in the lab. I performed iterative pruning, where I trained the full network to 0.9152 accuracy, pruned with s = 0.75 (my optimal starting point found before), then I retrained using the fine-tuning procedure (7 epochs), pruned the network again at s = 1.0, retrained using fine-tuning (7 epochs), and pruned a final time at s = 1.25 to achieve the sparsity I was aiming for (above 85%) and did the fine-tuning again (15 epochs). My thought here is that although I could have started at 1.25 and fine-tuned, that this gradual iterative technique would give the probability distribution more time to shift and thus overall keep more of the important weights. My theory held true (as seen below), because my final accuracy after pruning was 0.8995. Then I performed some testing on the quantization. In the quantization, I quantized at 5 bits, then used my function finetune\_after\_quantization() which is nearly the same as the pruning fine-tune function except it saved the net to a file “net\_after\_quantization.pt.” I performed one iteration of iterative quantization where I quantized to 5 bits, fine-tuned, and re-quantized (as my fine-tuning function did not zero the gradients between the centroids). By doing this I was able to achieve an accuracy of 0.8972. As this is rather close to the cut-off, I decided to not quantize further and see if I can achieve the specified compression ratio with this level of quantization. I then performed Huffman coding, performed the compression calculation you see below, and found I had met/exceeded the specified criteria so I decided to stop.
* Note, to check that the quantization occurred correctly, I created some code to access and depict the weight information of one layer. I then used a histogram with many more bins than quantized centroids. The discretization of the weight distribution told me that weights were being correctly shared (as seen below).
* Note, the final outputs are all saved within the DeepCompression.ipynb notebook as well
* Below is the calculation for the Huffman length:
  + Layer 1: (4.7788 \* 113) = 540 bits
  + Layer 2: (4.6694 \* 1107) = 5169 bits
  + Layer 3: (4.6401 \* 2856) = 13252.1 bits
  + Layer 4: (4.6302 \* 6190) = 28660.9 bits
  + Layer 5: (4.6285 \* 12272) = 56801 bits
  + Layer 6: (4.6752 \* 24806) = 115973 bits
  + Layer 7: (4.6237 \* 24098) = 111421.9 bits
  + Layer 8: (4.5612 \* 44970) = 205117.2 bits
  + Layer 9: (4.4531 \* 84209) = 374991.1 bits
  + Layer 10: (4.4547 \* 81276) = 362060.2 bits
  + Layer 11: (4.4317 \* 75491) = 334553.5 bits
  + Layer 12: (4.4503 \* 72493) = 322615.6 bits
  + Layer 13: (4.4675 \* 54521) = 243572.6 bits
  + Layer 14: (4.5048 \* 10625) = 47863.5 bits
  + Layer 15: (4.6289 \* 8450) = 39114.2 bits
  + Layer 16: (4.9011 \* 283) = 1387 bits
  + Average Encoding Length = sum(Layer Lengths)/total\_num\_param = 2263092.8/503760 = 4.49 bits
  + Memory Reduction (%): = 1 – average\_Huffman\_length/original\_length x 100% = 1 – 4.49/5 x 100% = 10.2% reduction in memory needed
* **Therefore, my final accuracy is 89.72>89.5 and my final compression rate is equal to 503760/3811680 x 5/32 x 4.49/5 = 0.01854. This constitutes a compression ratio of 1/0.01854 = 53.93x > 40x = 1/0.025. I have therefore, by my calculations met all the requirements and exceeded them by a fair margin.**

