Huffman Coding

Chih-Hsuan Chen, 110550029

***Abstract*—This experiment evaluates adaptive Huffman coding for compressing a neural network model file, alexnet.pth. Huffman coding and its adaptive variant with different symbol size, were implemented to assess compression efficiency. Performance was measured using average codeword length (or compression ratio) analysis. Results show that adaptive Huffman coding doesn't improve the performance.**

***Keywords*—Huffman code; compression; information; transition; entropy**

I. Introduction

Efficient data compression reduces storage and transmission costs, especially for large files such as models used in AI. This experiment applies Huffman coding to compress an ANN model *alexnet.pth*, comparing static and adaptive approaches for 8-bit to 32-bit data representations. Performance is evaluated based on codeword length (or compression ratio) analysis. The goal is to determine if adaptive Huffman coding enhances compression efficiency over standard methods

II. Implementation Details

A. *Static Huffman Algorithm*

The main concept of Huffman coding algorithm is using the last bit of codeword to differentiate the two (or more) symbols with least probability, and then combine them to a new symbol. It does this operation recursively to construct a tree – *Huffman Tree*, so that it can use the path on the tree to determine the codeword for each symbol and ensure it is a prefix code.

The supported symbol size of my coder is: 8-bit, 16-bit, 32-bit, 64-bit.

The flow of my implementation:

- Encode
  1) **Count the occurance and compute the probability for each symbol**
     Since static Huffman algorithm requires the probability information, the coder will pre-read the entire file and count the probability for each symbol at the beginning. The coder stores the occurance information with a multileveled table, which has 2^8 entries for each level, and stores the probability information in a *priority queue*.
  2) **Construct Huffman Tree**
     It is known that binary Huffman algorithm will do the combine operation *(num(symbols) - 1)* times, and generate a set (new symbol) each time, the coder uses an *unordered map* of size *num(symbols)* to store the symbols, and the another of size *(num(symbols) - 1)* to store the sets.
     The first element of these unordered map are the symbols, and the second is *tuple*s, whose first and second elements represent which set it belongs and its last bit, respectively.
  3) **Determine the codeword for each symbol**
     After constructing the code tree, the coder traverses it from leaf of each symbol to the root (the set which belongs to itself), to derive the codeword of each symbol. It also stores the codewords with their actual length in a multileveled table.
  4) **Output the code table and the final result**
     Before output the encoded data, the coder will output the code table and its size at the beginning of the file, it outputs *(symbol, length, codeword)* for each symbol. Then, it reads the input file and outputs the corresponding codeword for each symbol read.
- Decode
  1) **Parse the code table**
     The coder parses the code table and its length from the beginning of the file, to know the mapping of symbols and codewords.
  2) **Output the result**
     The coder decodes the remaining file and check the code table, and then output the original symbols.

B. *Adaptive Huffman Algorithm*

Since it's not possible to know the distribution of data source most of the time, the adaptive Huffman algorithm maintains a dynamic code tree rather than a static one. Both transmitter-side and receiver-side have to update their trees synchronously.

The supported symbol size of my coder is: 8-bit, 16-bit, 32-bit. (Note: 64-bit is not supported)

The flow of my implementation:

- Encode
  1) **Initialize the code tree**
     The coder allocates space for the root node and initialize its fields, and then Let NYT node pointer be equal to the root.
     Since each node has an unique ID in adaptive Huffman algorithm, the coder maintains two *unordered map*s: *id_table* and *node_table*. The first stores the mapping of symbols and their node IDs, and the second stores the mapping of node IDs and the corresponding node pointers.
     Also, there is another unordered map that stores weights and their blocks (the node IDs which have that weight).

2) **Check whether it's the first appearance or not**
For each symbol read, the coder checks *id_table* to determine whether its the first appearance of the symbol or not.
3) **Output the code according to the code tree**
For each symbol read, if it is the first appearance, the coder outputs the code tree path to NYT node followed by the symbol itself; otherwise, it outputs the path to the symbol's node.
4) **Update the code tree**
For each symbol read, the coder will increment the weight of the symbol's node and its parent (generate the new node from NYT node if it's the first appearance), and then switch the node with the node that has the larget ID in the same block (with the whole subtree). The coder does this operation recursively until it reaches the root node.

- Decode
1) **Initialize the code tree**
The procedure is the same as encoding.
2) **Receive the data and output the original symbol**
Every time the coder receive one byte, it reads the byte bit by bit, and traverses the code tree according to the bits read.
Once it reaches a leaf node, if that is the NYT node, the coder reads a symbol and output it directly; otherwise, it will output the symbol of that node.
3) **Update the code tree**
Every time the coder output a symbol, it updates its code tree as encoding.

After encoding and decoding, I used *diff* to compare to the original source file, to ensure the correctness of the coder.

III. Open-Source Usage

The source data file used in this experiment is an *AlexNet model*, which is a CNN architecture containing eight layers (five convolutional + three fully connected). It can recognize 1000 different types of objects given a 224*224 input image.

The key feature of AlexNet model:

1) **ReLU Activation**
Instead of Sigmoid or Tanh, AlexNet used ReLU (Rectified Linear Unit), which improved training speed by avoiding vanishing gradients.
2) **Overlapping Max-Pooling**
Instead of non-overlapping pooling, AlexNet used overlapping max-pooling to reduce overfitting and extract better spatial features.
3) **Dropout Regularization**
ntroduced dropout 50% in fully connected layers to prevent overfitting.
4) **Data Augmentation**
Used image translations, reflections, and PCA-based lighting changes to artificially increase the training dataset size.
5) **GPU Acceleration**
AlexNet was trained on two NVIDIA GTX 580 GPUs, splitting the network into two parallel processing streams.

IV. Experimental Results

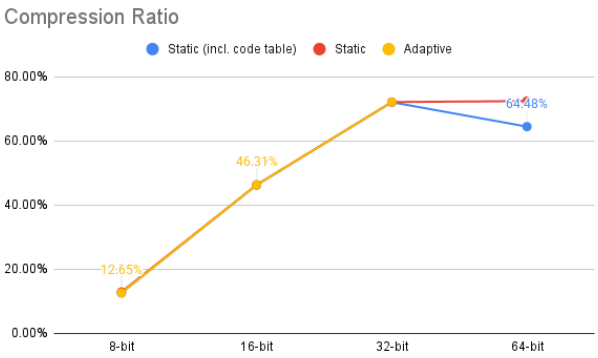

Fig. 1. Compression Ratio Comparison

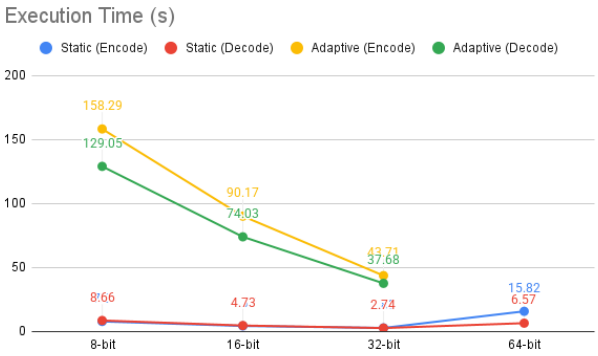

Fig. 2. Execution Time Comparison

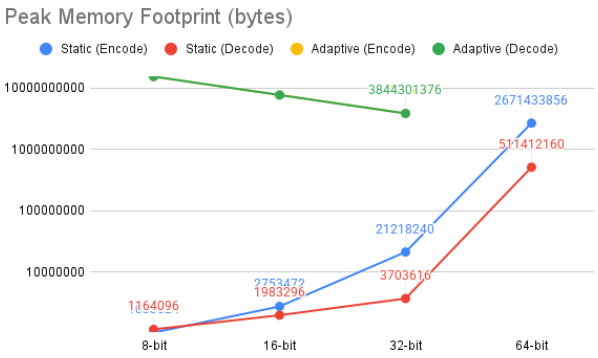

Fig. 3. Peak Memory Footprint Comparison

The following are the PMF charts of the entire file and the first four 40MB blocks.

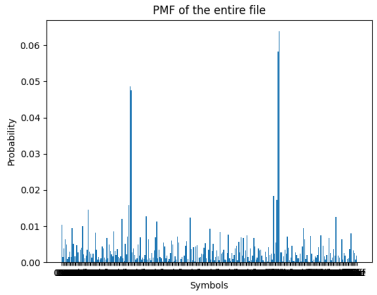

Fig. 4. PMF (8-bit) of the entire file

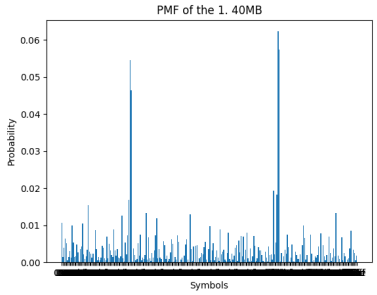

Fig. 5. PMF (8-bit) of the 1. 40MB

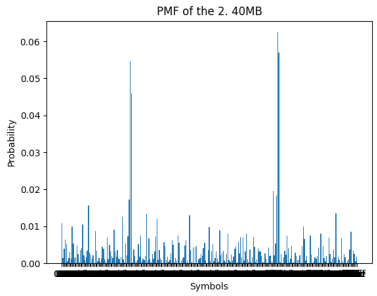

Fig. 6. PMF (8-bit) of the 2. 40MB

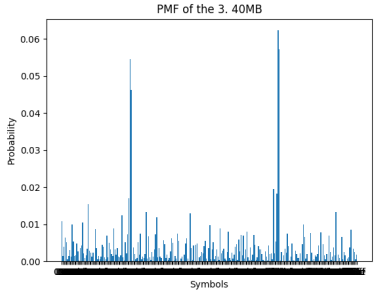

Fig. 7. PMF (8-bit) of the 3. 40MB

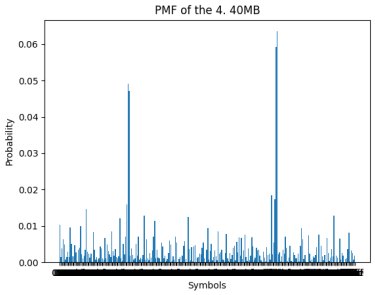

Fig. 8. PMF (8-bit) of the 4. 40MB

V. Discussions

From Fig. 1., we can find that the compression ratio of these two algorithms are almost the same, so the adaptive one doesn't improve the performance of static one in this case.

Basicly, the larger the symbol size is, the larger the compression ratio will be. However, the difference between compression ratio of 32-bit and 64-bit symbol size is very subtle, but the code table in the latter occupies a significant amount of file size.

Fig. 2. also shows that the larger the symbol size is, the shorter the execution time will be, while the execution time of static Huffman algorithm with 64-bit symbol size is longer than 32-bit. And the execution time of adaptive one is about 15 to 20 times slower than static one. Fig. 3. shows that their memory usages have the same trend.

To sum up, 32-bit symbol size will be the better choice for both algorithm. Although adaptive Huffman algorithm doesn't improve the performance of static one in this case and it consumes much more time and memory to execute, if the distribution of data source cannot be known in advance, this algorithm is still useful to compress the given information.

As for the PMF of symbols, Fig. 4.to 8. show that it doesn't change significantly throughout the whole file. (suppose the symbol size is 8 bits)

VI. Appendix

A. Environment

- Programming language: C++
- CPU: Apple M1 8-core (4 high-performance + 4 efficiency cores)
- Memory: 16GB
- Compiler: Apple clang version 16.0.0 (clang-1600.0.26.6)
- Measurement Tool: /usr/bin/time -al, diff

References

[1] VISO AI, *AlexNet Explained: A Deep Dive Into the Architecture*, https://viso.ai/deep-learning/alexnet/

[2] Ching-I, *CNN Classical Models: LeNet, AlexNet, VGG, NiN (with PyTorch code)*, https://medium.com/ching-i/%E5%8D%B7%E7%A9%8D%E7%A5%9E%E7%B6%93%E7%B6%B2%E7%B5%A1-cnn-%E7%B6%93%E5%85%B8%E6%A8%A1%E5%9E%8B-lenet-alexnet-vgg-nin-with-pytorch-code-84462d6cf60c