Analyzing stored AI models with Huffman Encoding

Authors Li-Ying, Hung, 311706012

***Abstract*—We evaluated two lossless data compression algorithms by analyzing statistical data distribution and evaluation metrics to assess their ability to compress AI models. We analyzed whether the algorithms can effectively reduce model storage space and improve the availability of model transmission. Finally, we believe that classic Huffman code is more suitable for compressing neural network models. Because the compression rates of the two are almost the same, but the encoding time of classic Huffman code is shorter.**

***Keywords*—data compression; neural network; huffman algorithm**

I. Introduction

In recent years, AI and machine learning have been frequently used to assist real-life applications. Among them, neural networks allow computers to automatically learn features from large amounts of data to generate human-like judgment capabilities. The biggest advantage is that the input and output of data are end-to-end, and only need to input the data into the model with well-trained weights. The output result is the inference result. Therefore, it simplifies the time for algorithm research and the difficulty of theoretical research. These powerful neural networks have begun to be integrated into daily life in commercial applications. However, there are still some challenges in real-life applications. Because AI models are stored as files and imported into memory when inference is needed. These models are usually getting larger and larger, and the largest known model size exceeds 1500B beyond TB level. Some remote server AI services will be transmitted through the network. We explored two lossless data compression algorithms, Huffman code and Adaptive Huffman code, to analyze whether the algorithm can effectively reduce model storage space and improve the availability of model transmission. The file is analyzed at the binary level and converted to statistical distribution, evaluate metrics such as compression ratio and expected length of source code. We compare the ability of both algorithms to compress AI models.

II. Related Work

The neural network (NN) which is the part of Artificial Intelligence (AI) is a popular research domain in nowadays. The neural network which is driven by a large amount of data can fit any mathematical function. As shown in Figure 1, The neurons are typically organized into multiple layers: input layer, hidden layer, and output layer. Each neural layer is composed of several neurons, single neuron is as follows:

$$X_4 = W_1X_1 + W_2X_2 + W_3X_3 + b$$

The output of the previous layer (L^0) is the input of the next layer (L^1) W is the weight, and b is the biases. In practical use, the open source machine learning library Pytorch will store the well-trained model contains all network structure, neuros weights and biases into *.pth* format file. These numbers are generally encoded 32-bit floating-point format by default in data file. Always have to call the entire model's weights and model structure into memory for use, when evaluating inference, or when remote server AI services call available APIs, We established an analysis experiment to study how to reduce model storage space and improve the availability of model transmission. More precisely, this kind of technology called data compression. Data compression is a technique that reduces the amount of data needed to represent a message or file. Data compression algorithms discuss source coding, which describes how to design an algorithm that maps symbols (sequences) from an information source to a code symbol set (usually bits). A well-designed data compression algorithm can minimize the amount of data for source symbols and accurately recover them from binary bits (lossless source coding) or with some distortion (lossy source coding).

Huffman code is a classic lossless data compression technique invented by American computer scientist David Albert Huffman in 1952. The algorithm first splits a set of source data into fixed-length bits, with one fixed-length bit representing a unique symbol, such as 256 kinds of symbols (2^8) for 8 bits as 1 symbol, and 65536 kinds of symbols for 16 bits as 1 symbol. Then the algorithm scans all of data organized by symbols to count the frequency of each symbol and builds a code tree from low to high frequency based on the symbol frequency, which is also called a Huffman tree. Symbols are coded into binary codewords according to their frequency of occurrence. The high-frequency symbols will be given shorter bits codewords to reduce space. Similarly, assume that 8 bits as 1 symbol, frequently used symbols usually have less than 8 bits, while less frequently used symbols consciously use longer length as codewords. The Huffman algorithm can be used to generate the smallest average expected length, and any other code for the same alphabet cannot have expected length lower than the code constructed by Huffman algorithm. However, Huffman code is not suitable for practical use:

- Inefficient for non-dyadic distribution
- Codeword length variations may be large
- Decoding complexity may be high, especially for CPUs

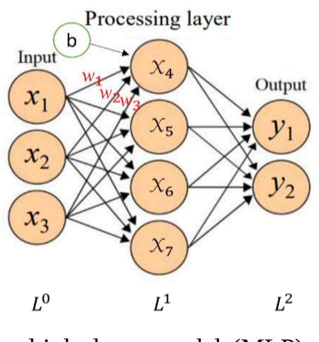


Figure 1: The multiple layer model (MLP) contains input layer, hidden layer, and output layer. And all layers are composed of neurons which are composed of weight and bias.

For example, in order to obtain the frequency of each symbol, the algorithm have to scan the entire data to obtain the distribution of the source data, and when encoding the second time, the message must be kept unchanged from the first time. Huffman code is not suitable for streaming video and instant messaging. Adaptive Huffman code is suitable for real-time transmission and can perform dynamic compression coding (also called Dynamic Huffman coding). It can encode, build a code tree and decode each symbol input at the same time. The algorithm has no initial knowledge of source distribution, so it updates the coding tree by statistically counting the occurrence of each symbol with each input symbol. This is similar to Huffman code, which moves frequently occurring symbols closer to the root of the tree to shorten their codeword length. Due to dynamic coding, the codeword for each symbol is not always fixed. Therefore, compared to classical Huffman code, Adaptive Huffman code has a lower encoding time cost for individual symbols and can dynamically adjust the code tree during the encoding process to achieve higher data compression rates. However, if we total the encoding data source cost, it requires more longer encoding time, computation and memory.

This research report is based on the analysis of classic Huffman code and adaptive Huffman code. We compressed AI model file by the above algorithms, and compared the feasibility, compression rate, and compression time evaluation between classic Huffman code and adaptive Huffman code.

III. Approach

This chapter discusses what the data we used for evaluation and which evaluation metrics was used to evaluate the quality of algorithms. We first introduce the source of AI models and model metadata, as well as distribution of the data. We investigate symbols composed of different lengths and compute the p.m.f. and the frequency. We made tables and charts that are easy to observe. Finally, we organize some metrics used to evaluate algorithms, including the average codeword length, the total running time of the algorithms, and batch comparison of processing time at 40MB, as well as the average processing time per symbol for algorithms.

A. Evaluation Data

We used the model format stored by the open-source machine learning library Pytorch in Python. It should be noted that the file only contains the weights and biases, not the model structure. More precisely, this file is a pre-trained model weight parameter. At inferring process, It is necessary to construct the model structure first, and then load the pre-trained weights from the stored file. The file size is 233MB. We use the model AlexNet as evaluation data. AlexNet is a classic convolutional neural network model structure and won the championship of the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012. Model architecture contains eight layers, the first five layers are convolutional layers, some of which are followed by maximum pooling layers, and the last three are fully connected layers. The statistical distribution is shown in Figure 2. We transform all kinds of bits length symbols to 256 reasonable and unified categories from observation. For detail, each category which have different sub kinds of symbols is shown in Figure 3. Then Figure 2 counted the frequency and storage space by each symbol happened in the entire file. As the bit length increases, frequency and size will appear more smoothly and closely to uniform distribution compared to baseline (8-bit symbol). Although frequency will decrease as the bit length increases, storage space by each symbol will have obvious fluctuations.

We count this AI weight file symbols in detail, between kinds of symbols that have appeared in the file and the theoretical value according to different bits as a symbol. When the file is treated 32 bits as a symbol, it has only 0.00017% of symbols existence. A lot of unused symbols may not good to design data compression. The column '*Kinds of symbols**' represents the kinds of symbols that have appeared in the file, while '*Kind of symbols*' represents the kinds of symbols that should exist theoretically inferred from the length of bits.

TABLE I. TYPES OF SYMBOLS STATISTIC

Bits	**Kinds of symbols***	**Kinds of symbols (Possible Combination)**	**Rate**
8	256	256	1
16	2875	65536	4.39%
32	7305	4294967296	0.00017%
64	1003725	18446744073709500000	-

We check whether the p.m.f. changes throughout the data source as shown in Appendix Table 4. We use 40MB as the default batch size and evaluate the variation of p.m.f. of each symbol by batch size, and calculate the average p.m.f., standard deviation, and symbol with maximum p.m.f. for each batch. The difference is that 'symbol with maximum p.m.f.' uses kinds of symbols instead of symbol category. We also evaluated bits composed of different lengths. The symbol with maximum p.m.f. is almost the same in different batches, but as larger bits situation, the average p.m.f. of batches tends to be uniform distribution, even if the symbol is the maximum p.m.f. it does not occupy large numbers of file.

B. Evaluation Metrics

- Expect length of source code ($E[l(xi)]$), let l(x) denotes the length of C(x), the expected length L(C) of a source code C(x) for a random variable X with p.m.f. p(x) is given by:

$$L(C) = \sum_{x \in \Omega_x} p(x) l(x)$$

- Data compression ratio is defined as the ration between the origin size and the compression size. If the ratio > 1 and the larger ratio, the smaller the data which is compressed. It means the compression algorithm can perform well to decrease the data size.

$$Compression\ Ratio = \frac{\text{Origin Size}}{\text{Compression Size}}$$

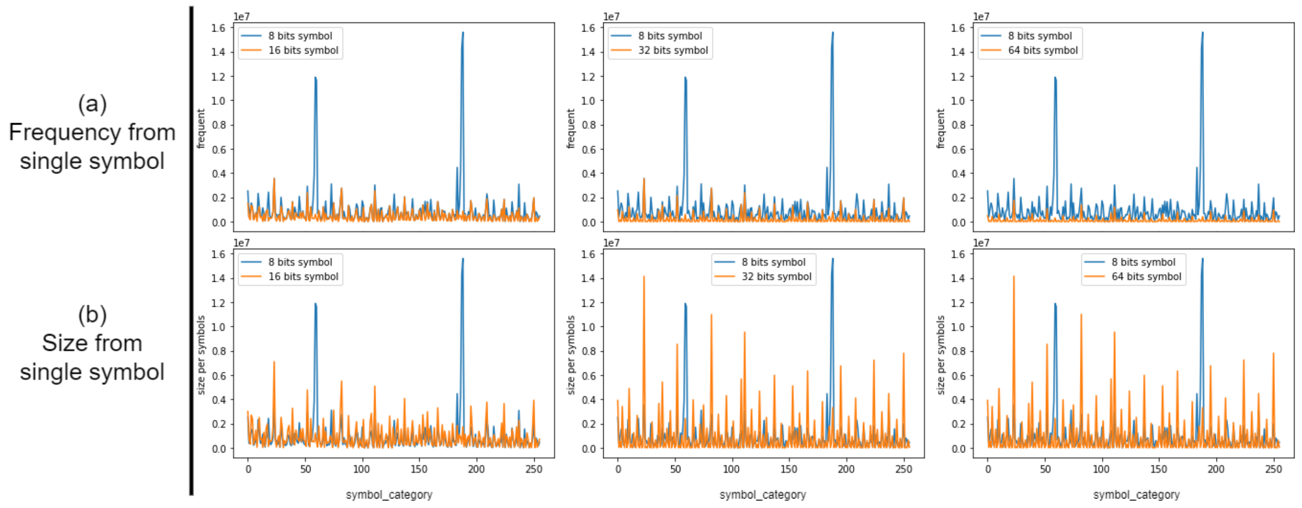


Figure 2: We compare the baseline (8 bits, blue line) with 16 bits, 32 bits, and 64 bits separately. (a) Single symbol's frequency is the frequency of different categories that appear in the weight file from 256 symbol categories (each category has numerous types of symbols). (b) Single symbol's size comes from each kinds of symbols' frequency multiply with different bits length, then total of them is the real size of each category.

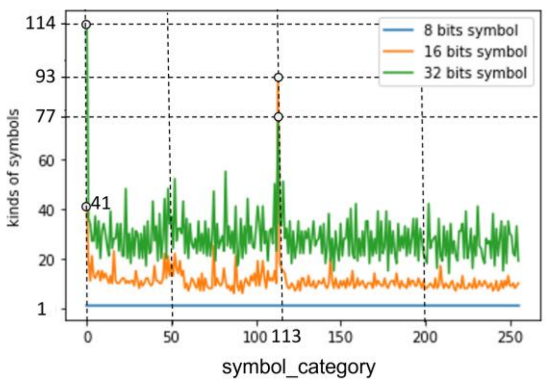


Figure 3: Because 8, 16, 32 bits as a symbol contain different kinds of symbols. We pick the symbol's first 8 bits as a category. When the bits length of sampling increase, every kinds of symbol's frequency will also increase. In 8 bits situation (blue line), shows that all of categories only have the fixed 1 kind of symbol; however, in 32 bits as a symbol, category 0 has 114 kinds of symbols variation. We highlight the AI model's weight data. Category 0 and 113 contain the most kinds of symbols at 16 bits, 32 bits situation.

IV. Experiments

According to Huffman code and adaptive Huffman code, we implemented these two compression algorithms in Python to analyze the compression rate of AI model's pretrained weight files. We read file in binary and treat specific bits length as different kinds of symbols. In the previous chapter, we analyzed the distribution of these symbols in the file. Since it is not easy to observe the difference between different bits length as symbols, we further divided the kinds of symbols into 256 symbol categories and separately counted the frequency and storage size of symbols appearing in the file. We also used batch analysis with a unit of 40MB and analyzed the p.m.f. changes of the total number of symbols in the entire file. These observation information are important information that affects the compression characteristics and advantages and disadvantages of algorithms. If a specific symbol appears most frequently but is uniform distributed in the file, it will affect the compression effect of Huffman code while adaptive Huffman code will have better compression rate.

We only implement and analyze the encoding part of algorithm, which does not include analysis of decoding after compression. Table 3 lists the performance comparison of Huffman code and adaptive Huffman code under different bits length as symbol: (1) How much compression rate can the algorithm encode the pretrained weight file? (2) The impact of different bits length symbols on compression rate. (3) Evaluate the performance of the algorithm with pretrained weight file in the average codeword length E[l(xi)]. In addition, we put the entropy into table as a baseline. (4) Comparison of the longest codeword length and the shortest codeword length. (5) The tree height of the Huffman binary tree.

A. *Algorithm comparison*

1) Huffman Code: As show in table 2. The total size of the original file is 244409199 Bytes. At different bits length as symbols, there are really close score between the baseline entropy and the average length of a code. Moreover, better compression rates can be achieved with longer bits lengths. In our implemented code, the situation of 8-bits symbol takes 387 seconds; 16 bits takes 4100 seconds; and 32 bits takes 2297 seconds. And it is noteworthy, the fact is that both 16 bits and 32 bits Huffman tree heights are greater than 8-bit tree height, and 8 bits is the smallest one in Codeword Length range and the average codeword length either. Another interesting part is that in Codeword Length range, the maximum value of 16 bits > 32-bits value > 8-bits value.

TABLE II. THE RESULT OF HUFFMAN ENCODING

Huffman Code						
Bits	**Total Size ↓**	**CR ↑**	**Entropy ↓**	***E*[*l*(*xi*)] ↓**	**CLR ↓**	**Tree Height ↓**
8	213493373	1.1448	6.96122	6.98806	4 ~ 15	16
16	112162146	2.1791	8.56537	8.59072	5 ~ 28	29
32	63602688	3.8427	8.89351	8.916345	7 ~ 26	27

*CR: Compression Ratio, CLR: Codeword Length Range
*Total Size(bytes), Entropy(bits/symbol), E[l(xi)] (bits/symbol)

2) Adaptive Huffman Code: As far as the results are concerned, the 8-bit symbol situation has a similar compression rate compare to classic Huffman code, also have the same average expected length. In the 16-bit symbol and 32-bit symbol situations, as our expected we analyze the data distribution before, there is a better compression rate than classic Huffman code, even though the codeword length range has doubled. This is because the codeword of this algorithm is different from that of classic Huffman code. When encoding a certain symbol, there are two situations: (1) unknown symbols will be arranged in the NYT position and added to their own bits. These two lengths are concatenated to form a codeword. (2) Known symbols will be encoded into the current symbol's position in the Huffman Tree. However, these processes will continuously update the tree, making the overall operation time of the algorithm extremely time-consuming. In our implemented code, 8-bit situation takes 25419 seconds (about 7 hours); 16-bit takes 95253 seconds (about 26 hours); and 32-bit takes 105688 seconds (about 29 hours). In addition, due to the dynamic encoding feature of the algorithm, we further record the average expected length of batch encoding, as shown in Figure 4. Perhaps because 40MB is still a considerable amount of data, there is not much variation in average expected length on different batches, and they are all very close to classic Huffman code.

TABLE III. THE RESULT OF ADAPTIVE HUFFMAN ENCODING

Adaptive Huffman Code						
Bits	**Total Size ↓**	**CR ↑**	**Entropy ↓**	***E*[*l*(*xi*)] ↓**	**CLR ↓**	**Tree Height ↓**
8	213493373	1.1448	6.96122	6.98806	~ 22	15
16	131235397	1.8624	8.56537	8.591192	~ 45	30
32	68132096	3.5873	8.89351	8.920397	~ 60	29

*CR: Compression Ratio, CLR: Codeword Length Range
*Total Size(bytes), Entropy(bits/symbol), E[l(xi)] (bits/symbol)

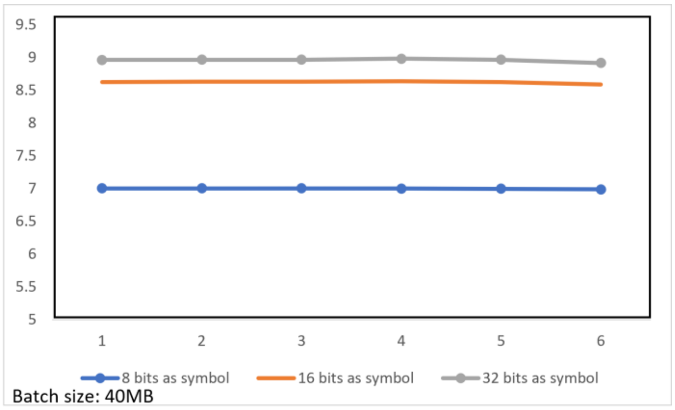


Figure 4: The average expected length of each batch recorded from the Adaptive Huffman encoding process.

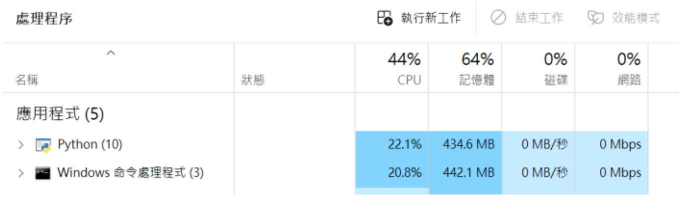
V. DISCUSSION

Although Adaptive Huffman code achieved better compression rates length as our expected and the average expected length was quite close to classic Huffman code, we believe that classic Huffman code would be a more suitable algorithm for compressing AI models compared in terms of tree update time and running time. In addition, there is a procedure to load neural network-based models, complete model weights must still be obtained before the model can be loaded. Maybe could not only compress whole pretrain weight file. We could think another way to store and load data for more efficient compression method. It seems that methods such as split the model into layers, then we could batch loading according to model layers or directly compressing the complete model file containing the model structure and pretrained weights, all can be discussed.

Another part that needs to be discussed is that this report still has some limitation. There are roughly three parts: (1) We only evaluated two algorithms. As we know, there are some similar algorithms such as Canonical Huffman code, Modified Huffman code, Extended Huffman code, etc., or explore other more powerful lossless compression algorithms. (2) Only alexnet model was used as a representative model, moreover, this model is only used for image classification tasks. Other different computer vision tasks such as semantic segmentation, object detection, etc. The different goals of model training may change the distribution of weights, which is also a direction that can be studied. (3) The implemented code of the classic Huffman code runs quickly in 8-bit symbol situations, but there is a big difference in speed between 16-bit symbol and 32-bit symbol situations. In addition, Adaptive Huffman code runs very slowly in all cases because Python itself runs slowly. We try to use third-party packages like numpy to accelerate and reduce the use of Python's for loop as much as possible. Two adaptive encoding process on CPU in Figure 5. It was not efficient shown from CPU usage rate. We plan to recheck the code and use JIT or Thread acceleration, and have a more unified benchmark as future work.

Final, the code has been published:

https://github.com/onlyin-hung/Analyzing-AI-Pretrained-Weights-with-Huffman-Encoding



名稱	狀態	44% CPU	64% 記憶體	0% 磁碟	0% 網路
應用程式 (5)					
Python (10)		22.1%	434.6 MB	0 MB/秒	0 Mbps
Windows 命令處理程式 (3)		20.8%	442.1 MB	0 MB/秒	0 Mbps

Figure 5

REFERENCES

[1] B. Shivali, “Huffman Coding Implementation in Python with Example | FavTutor, ” , January 2022. https://favtutor.com/blogs/huffman-coding

[2] V. Rohan, “DataEncoding- Adaptive Coding, ”, April 2019. https://github.com/HEXcube/DataEncoding/tree/master

[3] T. Chun-Jen, “Huffman Coding, ”, February 2014. https://people.cs.nctu.edu.tw/~cjtsai/courses/imc/classnotes/imc14_03_Huffman_Codes.pdf

[4] H. HAO-JIE, “Adaptive Huffman Encoding Example Practice,”, March 2017, https://www.youtube.com/watch?v=N5pw_Z-oP-4.

APPENDIX

TABLE IV. BATCH RESULT

8 bits as a symbol							
Batch size: 40 MB	**0**	**1**	**2**	**3**	**4**	**5**	**All file**
Average	0.00067	0.00067	0.00067	0.00067	0.00067	0.00055	0.00391
Std	0.00125	0.00125	0.00125	0.00123	0.00125	0.00097	0.00716
Max(symbol_idx)	[187]	[187]	[187]	[188]	[188]	[188]	[188]
Max(p.m.f.)	1.0696%	1.0718%	1.0698%	1.0899%	1.2570%	1.0892%	6.3808%
16 bits as a symbol							
Average	0.00010	0.00010	0.00010	0.00010	0.00010	0.00008	0.00035
Std	0.00031	0.00031	0.00031	0.00029	0.00028	0.00019	0.00134
Max(symbol_idx)	[23 183]	[23 183]	[23 183]	[23 183]	[23 183]	[23 183]	[23 183]
Max(p.m.f.)	0.5277%	0.5341%	0.5297%	0.5009%	0.4930%	0.3040%	2.8894%
32 bits as a symbol							
Average	0.00010	0.00010	0.00010	0.00010	0.00010	0.00008	0.00014
Std	0.00021	0.00021	0.00021	0.00020	0.00019	0.00012	0.00059
Max(symbol_idx)	[0 0 0 0]	[0 0 0 0]	[0 0 0 0]	[0 0 0 0]	[0 0 0 0]	[0 0 0 0]	[0 0 0 0]
Max(p.m.f.)	0.1484%	0.1522%	0.1500%	0.1418%	0.1416%	0.0831%	0.8171%

*From the first batch to the fifth batch are independent compute their p.m.f.. The last “All file” compute from the whole file.