Huffman Coding 的實驗分析

張君實, 0810749

I. Introduction

近年 AI 非常熱門，也有許多 AI 模型問世，而 AI 模型的大小通常都不小，對 Transformer 來說，大多 transformer 都是 GB 或 TB 量級，而 AlexNet 就是一個較小的模型，大概有 240MB。為了處理過大的模型，壓縮演算法的使用變的非常的重要。一提到壓縮演算法，Huffman Coding 應該是其中最有名的壓縮演算法。就理論的層面而言，Huffman Coding 的 Coding Length 可以達到最佳化。而在這次的作業中，我們必須實作 Huffman Coding 和 Adaptive Huffman Coding 壓縮 AlexNet，並且能解壓縮 AlexNet 回來。此外，我們還要分析不同壓縮演算法的效率。

II. Related work

Huffman coding [1] 是一個很有名的壓縮演算法，其演算法在 Huffman 讀博班時被提出，此外，Huffman 也證明 Huffman Coding 在 D - Adic 時，其期望 Coding Length 可以達到 Entropy。而雖然 Basic Huffman Coding 在 Coding Length 上效果不錯，但是它有不少的缺點。第一點，它沒有考慮前後 Symbol 的關係。二來它只能處理 Offline 的任務，也就是說，在面對一個 Online 的任務時，Basic Huffman Coding 只能等任務結束後才能進行壓縮。三來，傳統的 Huffman Tree 並沒有考量到樹的平衡，也就是說，對於某些的 Symbol 可能會有過長的 Coding。而為了解決上述的三個缺點。就產生了 Adaptive Huffman Coding [2]，Extended Huffman Coding 和 Length-Limited Huffman Coding [3]。

為了處理 Online 的任務，Adaptive Huffman coding 演算法因此誕生。關於 Adaptive Huffman Coding，主流的演算法分成 FGK 演算法和 Vitter 演算法，這兩個演算法各有優缺點，FGK 演算法每更新一個節點時，最多只需要交換一次節點，在壓縮速度上較快。而 Vitter 演算法每更新一個節點時，可能會交換很多次節點，壓縮速度較慢，但是長出來的 Huffman Tree 會更加平衡，在這個作業中，為了速度考量，我所採用的 Adaptive Huffman Coding 演算法為 FGK 演算法。

由於 Basic Huffman Coding 每一個 Symbol 只包含了一個 Character，這意味著 Basic Huffman Coding 並不會考慮 Character 間的前後文關係。但是常常一個檔案中，Character 間是有前後文關係的，像說 a 後面接的機率 e 比 e 後面接 a 的機率小很多。為了考慮 Character 間的前後文關係，於是出現了 Extended Huffman Coding 演算法，也就是每一個 Symbol 包含了好幾個 Character。因為每個 Symbol 包含了好幾個 Character，自然就考慮了前後文關係。

此外，不平衡的 Huffman Tree 也不是大家願意看到的情況，假如有 Huffman Tree 太長，有可能部分字元 Encode 時反而比沒 Encode 前長，若剛好又使用了 Extended Huffman Coding，使得 symbol 數又特別的多，太長的 Codeword 對於 Huffman Coding 的 Header 設計來說就充滿了挑戰性。因此，Length-Limited Huffman Coding 就誕生了，Length-Limited Huffman Coding 透過了 Package-Merge Algorithm，使得 Huffman Tree 看起來相對平衡許多。

III. Method

A. *What I finish in the homework*

在本次作業中，我實作了 8-Bit Huffman Coding，32-Bit Huffman Coding，8-Bit Adaptive Huffman Coding 和 Extended Huffman Coding。

B. *The implementation detail of Basic Huffman Coding and Extended Huffman Coding*

有關 Basic Huffman Coding 和 Extended Huffman Coding，我基本上是實作在一起，因為 Extended Huffman Coding 不過就是每個 Symbol 包含超過 8Bit 的 Basic Huffman Coding。為了下文的稱呼方便，我將 Basic Huffman Coding 和 Extended Huffman Coding 統稱為 Huffman Coding。為了實作上方便，我在寫 Huffman Coding 時，是使用 python 來實作。

演算法的部分，大家應該耳熟能詳，基本上就是對每個 Symbol 分別建一個只有 1 個節點的 Tree，而 Tree 的 Root 存放著每個 Symbol 頻率的總和。Huffman Coding 的過程就是每次將兩個最少 Symbol 頻率和的 Tree 合併在一起，直到合併到剩下一個 Tree 為止，最後再從 Root Traverse 到 Leaf 獲得每個 Symbol 的 Encoding。

接著介紹我的壓縮檔格式(圖 3-1)，首先我的前 32Bit 是儲存 Table 的 Symbol 數量。而由於 Extended Huffman Coding 每次都是讀至少 2 個 Character，這就意味著最後可能會有一些 Character 被多餘出來，而多餘出來的 Character 因為也只會出現一次，拿去 Huffman Tree 中壓縮也只是浪費空間，因為 Table 就會表示一次多餘的 Character，所以在實作上，多餘出來的 Character 我的處理是不壓縮。因此，我的 33~64Bit 是儲存多餘的 Character 長度。65Bit 以後直接放上多餘的 Character。

處理完多餘的 Character 後，接下來就建 Table，我建 Table 的方法，就是將所有 Symbol 和對應的 Symbol Frequency 寫入 Table。而 Symbol 花多少 bit 存，就看

Huffman Coding 設定多少 Bit 存取。而對應的 Frequency，我是使用 32Bit，也就是 Integer 的格式存取。

處理完 Table 後，接下來放置編好的 Encoding Sequence，但是因為 Encoding Sequence 可能不是 8-Bit 的倍數，可是檔案都得用 Byte 來存，所以 Encoding Sequence 要在後面補一些多餘的 Bit，使其成為 8Bit 的倍數。而因為補了多餘的 Bit，我們為了知道我們補了多少 Bit，所以我們多花了 8Bit 在壓縮檔的最後一位，使得我們知道我們補了多少 Bit。

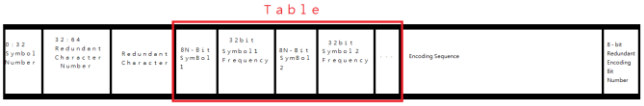

圖 3-1: Huffman Coding 壓縮檔格式

Decoding 的過程也很簡單，首先先讀前 64bit 和最後 8bit 分別獲得 Table 中 Symbol 的數量，多餘的 Character 長度和多餘的 Encoding Bit 長度，接著透過剛剛獲得的資訊把多餘的 Character 和 Table 建出來，有了 Table 就可以建樹和把 Encoding Sequence 給 Decode 回來，最後再將多餘的 Character 接在 Decoding Sequence 後，一切就解壓縮完了。

至於 Symbol 總共有幾 Bit，這個我直接寫進了壓縮檔的副檔名內，而 Huffman Coding 壓縮檔的副檔名我取名為.hcps[num]，[num]是一個數字代表幾個 byte，所以如果副檔名為.hcps1 就代表是 8-Bit Huffman Coding，如果副檔名為.hcps4 就代表是 32-Bit Huffman Coding。

C. The implementation detail of Adaptive Huffman Coding

有關 Adaptive Huffman Coding 的實作，因為他是邊讀取邊更新 Tree，很吃程式的效率性。再加上我看到我用 python 實作的 64-Bit Huffman Coding，花了一整天，還是沒辦法把 100 萬個節點壓成一棵 Tree，所以 Adaptive Huffman Coding 我是用 C++實作的，且為了方便管理所有 Node 的編號，我只有實作 8-Bit Adaptive Huffman Coding。

Adaptive Huffman Coding 主要分成 FGK 演算法和 Vitter 演算法，而在效率上來說，FGK 演算法比 Vitter 演算法快一些，因為 FGK 演算法在每次更新一個節點時，只會和相同權重中順序最大的節點交換，而 Vitter 演算法在每次更新一個節點時，會和相同權重中所有順序大的節點交換一遍。當然 Vitter 演算法的好處在於，他生成的 Tree 會更佳的平衡。而在實作中，為了效率考量，我選擇使用 FGK 演算法。

接著仔細講解 FGK 演算法。首先是壓縮的演算法，都是每讀一個 Symbol，更新一次 Tree，然後再輸出對應的 Encoding 或 Header。

先講怎麼更新樹的，在還沒讀 Symbol 前，我們要建立一棵只有節點 NYT 的 Tree，之後每讀到一個 Symbol，如果 Symbol 先前沒有出現過，就將原 NYT 的位置長出兩個分支，其中一個是新的 NYT 位置，另一個是放 Symbol，然後分支權重都初始化為 0，分支的順序分別為原 NYT 節點減 1 和減 2。接著就要從 Symbol 該節點做更新，每次更新都是和同權重中比最大順序的節點交換，當然如果最大順序的節點是自己就和自己交換。交換完以後就將自己權重加 1，接著遞迴至父節點，重複上述更新的過程直到根節點。

接著講如何生成對應的壓縮檔，假如 Symbol 第一次出現，則是輸出當前 NYT 的 encoding 和 Symbol，若 Symbol 不是第一次出現，則輸出 Tree 更新前 Symbol 的 Encoding。假如最後的壓縮檔不是 8Bit 的倍數，直接在後面補 NYT 節點的部分 encoding，再補不滿就補 0 到 8Bit 就行。

壓縮講完了，接著講 Adaptive Huffman Coding 如何解壓縮。基本上，就是一邊讀壓縮檔，另一方面去走 Tree 的路徑並更新樹，並輸出 Decoding Sequence。

假如一邊讀壓縮檔中的 Encoding，另一方面去走 Tree 走到了節點 NYT，接下來讀的 8Bit 就是 Symbol，所以讀完了 Symbol，就要照著前面提到更新樹的方法下去更新樹，並輸出 Symbol。如果透過 Encoding 走到了節點其他 Symbol，也是要照著前面提到更新樹的方法下去更新樹，並輸出 Symbol。

而先前提到壓縮檔不是 8-Bit 的倍數的做法，在處理最後 8-Bit 時是安全的，我們可以分成 2 個情況，第一種是 Tree 沒有走到 NYT 節點，第二種是 Tree 有走到 NYT 節點。第一種情況顯然不會對解壓縮檔有任何輸出，而第二種情況，就算後面隨便補 0，因為補的 0 數量一定不滿 8bit，但是 Symbol 都是 8-Bit，所以也不會有對應的 Symbol 會被 Update。因此上述的邊界處理是安全可靠的。

IV. EXPERIMENT

A. The expected coding length and size

在這個實驗中，我們會比較 Huffman Coding，Adaptive Huffman Coding 和 Extended Huffman Coding 之間的期望 Coding Length 和實際壓縮的大小。

Algorithm	Expected Coding Length
8Bit Huffman Coding	6.9881 Bits
16Bit Huffman Coding	8.5907 Bits
24Bit Huffman Coding	12.0824Bits
32Bit Huffman Coding	8.9163 Bits
8Bit Adaptive Huffman Coding	6.9881 Bits

表 4-1 演算法與對 Expected Coding Length 關係

首先我要說明 Expected Coding Length 是怎麼計算的，我這裡是以 Symbol 為單位取期望值，不是以 Bits 為單位取期望值，我個人認為從數學的角度來說，用 Symbol 為單位取期望值比較合理，且比較能進行理論分析。從表 4-1 可以看出，對於越多 Bits 的 Symbol，Expected Coding Length 也會普遍變長。這其實很直覺，Huffman Coding 的 Expected Coding Length 近乎於 Entropy，原文書有提到說 Entropy 的上限是符號的總數取 Log，而當 Symbol 為 8Bit 時，Symbol 總數有 256 種，當 Symbol 為 32Bit 時，這個檔案中出現的 Symbol 總數總共有 7000 多種，所以 Expected Coding Length 隨 Bit 長度增加是很直觀的事。

而從表中也能觀察到，24Bits Huffman Coding 的 Expected Coding Length 是裡面最長的，其原因在於說，大多資料都是以 8 或 16 或 32Bits 存取，所以如果用 24Bits 來做 Huffman Coding，有很大的可能拿到的資料沒有上下文關係，可能比較近於 IID 性質，那 Entropy 自然就高。

此外，從這次實驗中，也可以注意到不管是 Huffman Coding 還是 Adaptive Huffman Coding，在檔案夠大的情況下，他們的 Expected Coding Length 基本上會收斂到差不多的位置。

Algorithm	Header Size
8Bit Huffman Coding	1289 Bytes
16Bit Huffman Coding	17254 Bytes
24Bit Huffman Coding	1543089 Bytes
32Bit Huffman Coding	58444 Bytes
8Bit Adaptive Huffman Coding	256 Bytes

表 4-2 演算法與壓縮檔 Header Size 關係

表 4-2 中可以看到，Header Size 隨 Symbol 的總數增加而增加。而值得注意的是， Adaptive Huffman Coding 因為可以在更新 Tree 時就直接知道 Encoding，所以其 Header 會普遍小於 8Bit Huffman Coding。

Algorithm	Total Size
8Bit Huffman Coding	213494662 Bytes
16Bit Huffman Coding	131245413 Bytes
24Bit Huffman Coding	124587312 Bytes
32Bit Huffman Coding	68159596 Bytes
8Bit Adaptive Huffman Coding	213494623 Bytes

表 4-3 演算法與壓縮檔 Total Size 關係

Algorithm	壓縮前後比值
8Bit Huffman Coding	87.3513 %
16Bit Huffman Coding	53.6990 %
24Bit Huffman Coding	50.9749 %
32Bit Huffman Coding	27.8875 %
8Bit Adaptive Huffman Coding	87.3513 %

表 4-4 演算法與壓縮前後比值關係

表 4-3 和表 4-4 可以一起看，可以發現大致上壓縮前後比值是有隨 Symbol 的 Bits 數增加而減少的。而比較驚人的地方在於，24Bit Huffman Coding 的 Expected Coding Length Per Symbol 雖然很長，但是若將其轉成 Expected Coding Length Per Bits 後，還是比 16Bits 的 Expected Coding Length Per Bits 還短，所以 24Bits 的壓縮前後比值就比 16Bits 的壓縮前後比值還低。從這個觀察可以告訴我們說，就算沒有任何理由的增加 Symbol 的數量，實務上的壓縮效果可能還是不輸較短 Symbol 的 Huffman Coding。

此外，從表 4-1 到表 4-4 來看，整體上 Huffman Coding 和 Adaptive Huffman Coding 壓縮後的表現幾乎沒有差別，可能 Huffman Coding 速度快了一些，但 Adaptive Huffman Coding 他可以處理 Online Task，所以還是依照使用場景決定使用哪個演算法比較好。

B. The distribution of the data source

Iteration\rank	1	2	3	4	5
0~244MB	188	187	59	60	183
0~40MB	187	188	59	60	183
40~80MB	187	188	59	60	183
80~120MB	187	188	59	60	183
120~160MB	188	187	59	60	183
160~2000MB	188	187	60	59	183
200~240MB	188	60	187	59	183
240~244MB	188	60	189	61	187

表 4-5 不同區間最高出現機率的 Symbol 排名表(1)

Iteration\rank	6	7	8	9	10
0~244MB	186	58	23	237	73
0~40MB	186	58	23	237	73
40~80MB	186	58	23	237	73
80~120MB	186	58	23	237	73
120~160MB	186	58	23	73	237
160~2000MB	186	58	23	73	237
200~240MB	186	58	23	73	111
240~244MB	59	183	52	111	23

表 4-6 不同區間最高出現機率的 Symbol 排名表(2)

表 4-5 與表 4-6 為各區間出現最多次的 ascii 的前 10 名，為了方便計算，這裡我將 1MB 當作 10^6B。我們可以看到，除了 200~240MB 與 240~244MB 這兩個區間的前十名 Symbol 跟其他區間不太一樣以外，其餘的區間分布大致相同。這就說明著，我們未來若想進一步壓縮這個檔案，我們可以先壓縮前 200MB，接著再壓縮剩下的 200~244MB。看起來還能進一步再壓縮下去。

V. CONCLUSION

從這次的作業中，我除了學到如何分別用 Python 和 C++來寫 Huffman coding 和 Adaptive Huffman coding。還有機會用上課學到的理論和實務的角度分析差異，這份作業算是讓我收穫不少。未來有時間的話我也會想寫 D-ary Huffman Coding 和 Length-Limited Huffman Coding，順便當作練習演算法。

REFERENCES

[1] D.A. Huffman, "A method for the construction of minimum-redundancy codes", Proceedings of the I.R.E., sept 1952, pp 1098-1102

[2] J. S. Vitter, "Design and Analysis of Dynamic Huffman Codes", Journal of the ACM, 34(4), October 1987, pp 825–845.

[3] Hirschberg, Daniel S. (1990). "A fast algorithm for optimal length-limited Huffman codes". Journal of the Association for Computing Machinery. 37 (3): 464–473.