Arithmetic Coding 的實驗分析

張君實, 0810749

I. INTRODUCTION

AlexNet 是一個非常有名的類神經網路模型，雖然已經過時，但是其使用 Maxpooling，ReLU 和 Dropout，仍是深度學習中的基礎且經典的技巧。在第一份報告中，我實作了各種 Huffman Coding 的演算法，來壓縮 AlexNet。

雖然說 Huffman Coding 理論上是最佳的編碼方法，實務上仍然有缺點。若 Huffman Coding 不滿足 D – Adic，期望 Coding Length 會比 Entropy 還大，因為每個 Symbol 的 Coding Length 只能取整數。而這又導致了一個問題，若符號的機率分布是傾斜的，也就是說有些符號出現次數特別多，就會導致期望 Coding Length 特別長。這時使用 Extended Huffman Coding 降低符號出現次數或許有用，但會導致另一個問題，Codebook 會太大，所以增加 Symbol 的數量不是總是有效。

為了改善 Huffman Coding 的缺點，可以改使用 Arithmetic Coding，雖然 Arithmetic Coding 理論上不是最佳的編碼方法，但是對於單一 Symbol 的平均 Coding Length，可以不是整數，這就能解決先前提及 Huffman Coding 的缺點和問題。而在本次報告中，我會實作各種 Arithmetic Coding 的演算法，並且分析與 Huffman Coding 間效率的差異。

II. RELATED WORK

Arithmetic Coding 是一個很有名的壓縮演算法，理論在 1960 年代早期被 Elias 提出[1]，實作則是 Rissanen 在 1970 年代提出[2]。與 Huffman Coding 不同的是，Huffman Coding 對於每個 Symbol，都會輸出固定的 Codeword。而 Arithmetic Coding 對於每個 Symbol，會更新相對應的區間，這也使得 Arithmetic Coding 輸出的 Codeword 不固定。

Arithmetic Coding 藉由更新小數區間來編碼，理論上方法可行。實務上若不多作處理，會遇到問題。由於 Arithmetic Coding 更新小數區間的方法可能會受到精度和位元數的限制，所以實務上會將更新區間的方式更改成整數運算。而 Arithmetic Coding 用整數運算更新區間時，區間會越來越小，所以也會引入 Rescaling 的機制，將區間拉大。

Arithmetic Coding 可以處理 Offline 的任務，如果要處理 Online 的任務，可以將 Arithmetic Coding 更改成 Adaptive Arithmetic Coding，其精神與 Adaptive Huffman Coding 類似，對於每次 Symbol 的輸入，同時進行編碼和更新 Table。

先前所述，Arithmetic Coding 可以有效處理 Symbol 機率分布傾斜的問題。若能讓 Symbol 機率分布越傾斜，Arithmetic Coding 就有越好的表現。而 PPM 演算法[3]藉由建立大量含有條件機率的 Table，使得 Symbol 的機率分布盡可能的傾斜，以達到更好的壓縮效果。

PPM 演算法雖然能盡可能的傾斜機率分布，但是仍有 Symbol 頻率太少，造成 Symbol 區間過小，Coding Length 過長的問題。如果是 Escape Symbol，對於 PPMA 的話，每個 Context 的 Escape Symbol 頻率都是 1，而 PPM Method B 和 PPM Method C[4]藉由查看 Context 中 Symbol 的數量，動態調整 Escape Symbol 的頻率，使得 Coding Length 不會過長。而對於一般的 Symbol，也可以引入 Exclusion Principle 的技術，當在 High-Order Context Escape 時，直接移除 Low-Order Context 中不可能出現的 Symbol，以此增加 Symbol 的區間，減少 Coding Length。

III. METHOD

A. *What I finish in the homework*

在本次作業中，我使用 Python 實作 Arithmetic Coding 系列的 Encoder。Arithmetic Coding 的部分，我使用整數區間分別實作 256 Symbol 和 Binary Symbol 兩種方式。對於更新 Table 的方式，我也實作了 Fixed Probability Model 和 PPM。而 PPM 演算法的 Order 範圍可以從 0 至 2，所以會有 8 種組合。

B. *The implementation detail of Arithmetic Coding and Table*

接著我要詳細敘述我如何實作 Arithmetic Coding 和 Table。

首先先討論 Arithmetic Coding，雖然我有實作 256 Symbol 和 Binary Symbol 兩種方法，不過 256 Symbol 和 binary Symbol 只差在讀取是一次讀一個 Byte 還是一次讀一個 Bit，背後 Arithmetic Coding 的原理還是大同小異。對於 Arithmetic Coding 的演算法，這次作業要求用整數區間實作，其完整演算法如同圖 3-1 所述。

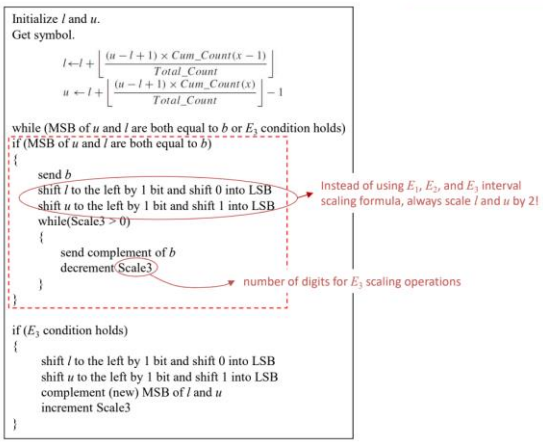

圖 3-1 Arithmetic Encoding 演算法流程

從圖一可以看到，每次收到一個 Symbol 後，就要更新一次區間，並且判斷是否滿足 E1/E2/E3 Scaling。如果是 E3 Scaling， 就要將[0.25, 0.75]的區間拉成[0, 1]，並且要增加 Scale3 的數目。如果是 E1/E2 Scaling，就是要將原本[0, 0.5]或[0.5, 1]的區間拉成[0, 1]，並且 Encode 一個 0/1 和 Scale3 個 1/0 到壓縮檔內，並讓 Scale3 歸零。

有關更新區間的公式，為了保持整數實作，所以更新 Upper Bound 和 Lower Bound 時，每次都要無條件捨去。而為了保證每次無條件捨去後，Decoder 有辦法解的回來，Upper Bound 和 Lower Bound 的 Bits 數至少要是 Ceiling(log N/ log2) + 2， 其中 N 為輸入檔案的 Symbol 數量。以這次作業 AlexNet 為例，其檔案大小為 244409199 Bytes，如果是每讀一個 Byte 就當一個 Symbol，依照公式 Upper Bound 和 Lower Bound 要至少 30Bits 表示，而如果是每讀一個 Bit 就當一個 Symbol，依照公式 Upper Bound 和 Lower Bound 要至少 33Bits 表示。為了實作方便，所以我在做 256 Symbol 時，我將 Upper Bound 和 Lower Bound 用 32 Bits 表示，而在實作 binary Symbol 時，我將 Upper Bound 和 Lower Bound 用 64 Bits 表示。

而當所有 Symbol 都讀完後，為了能 Decode 回來，要加上 EOS 的符號，而 EOS 的符號可以任意選擇 Upper Bound 和 Lower Bound 之間的任意數字，我的做法是直接將最終的 Lower Bound 當作 EOS 符號。除了 EOS 的符號外，因為最終編碼序列可能不是 8 Bits 的倍數，所以我又多花 8 Bits 表示 Padding 的數量，並寫在 EOS 符號前。因為我們事前知道 EOS 的長度，所以也就知道表示 Padding 數量的 8 Bits 的位置。

講完 Arithmetic Algorithm 的詳細流程後，接下來講解 Fixed Probability Model 與 PPM 的演算法與實作。Fixed Probability Model 的意思就是透過估計資料的 Symbol 機率分布，建立相對應的 Table 供 Arithmetic Algorithm 使用。而我 Fixed Probability Model 的做法很單純，就是將整個輸入檔讀一遍，統計每個 Symbol 對應的頻率，建立一份對應的累積頻率表。

至於 PPM 的部分，一個 Table 會包含不同的 Order，不同 Order 又會包含不同 Context 作為條件。PPM 的演算法如下所述，每次讀進新的 Symbol，都從最高的 Order 開始尋找對應的 Context，如果找不到對應的 Context 就跳過，如果找到了對應的 Context 條件，但是 Symbol 的頻率卻是 0，這時就要輸出 Escape Symbol。遇到上述兩種情形，接著都要到較低的 Order 重複上述的動作，直到 Context 中的 Symbol 頻率不為 0 為止。若當 Order 為 0 時，還是輸出 Escape Symbol，就會去 Order 為-1 的表中找 Symbol，而 Order 為-1 的表中每個 Symbol 的頻率相同。等到輸出結束後，接著就是對於輸入的 Symbol 去更新 Table，所有 Order 和對應的 Context 中的 Symbol 頻率都要加一。

接著講 PPM 實作的部分。整個 Table 我是用 Tree 的結構下去建的，Tree 的 Root 存放 Order = 0 的頻率表，每往下一層，其 Node 的 Order 就加一。所以對於 256 Symbol 來說，如果每種 Context 的組合都有出現，每個 Node 就會有 256 個 Children。對於 Binary Symbol 而言，如果每種 Context 的組合都有出現，每個 Node 就會有 2 個 Children。而透過 Traverse 的方式，就可以找到對應的 Context 的頻率表做計算與更新。

PPM 實作也有許多細節可以討論，像是 Escape Symbol 的頻率，因為我用 Python，怕效率不佳，我採用 PPMA 的做法，每個 Context 中的 Escape Symbol 的頻率為 1。而基於速度考量，我也沒使用 Exclusive Principle。至於有關 Order 的實作，我設定成 Order 是正整數和 0 都能執行。不過為了效率考量，實驗中我只會附上 Order 從 0 至 2 的結果。

IV. EXPERIMENT

A. The expected coding length and size

在這個實驗中，我們會比較 8Bit Huffman Coding，256-Symbol AC+FPM， Binary AC+FPM， 256-Symbol AC+PPM 和 Binary AC+PPM 之間的期望 Coding Length 和實際壓縮的大小。

Algorithm	Expected Coding Length
256-Symbol Huffman Coding	6.9881 Bits
256-Symbol Adaptive Huffman Coding	6.9881 Bits
256-Symbol AC + FPM	6.9612 Bits
256-Symbol AC + PPM Order = 0	6.9612 Bits
256-Symbol AC + PPM Order = 1	2.8427 Bits
256-Symbol AC + PPM Order = 2	2.2606 Bits
Binary AC + FPM	0.9992 Bits
Binary AC + PPM Order = 0	0.9992 Bits
Binary AC + PPM Order = 1	0.9990 Bits
Binary AC + PPM Order = 2	0.9976 Bits

表 4-1 演算法與對 Expected Coding Length 關係

對於 Expected Coding Length 的計算方法，我是對壓縮檔整體大小扣除 Header 大小後，再除 Symbol 的數量。所以 Binary Arithmetic Coding 的 Expected Coding Length 明顯小於 256-Symbol Arithmetic Coding 是正常的，因為 Binary Arithmetic Coding 的 Symbol 總數是 8 倍。

接著我們比較 256-Symbol Arithmetic Coding + FPM 和 256-Symbol Huffman Coding 的 Expected Coding Length，我們發現實務上確實 256-Symbol Arithmetic Coding 效果會比 256-Symbol Huffman Coding 好一點，其原因在於 Huffman Coding 對於每個 Symbol 的編碼都是整數長度，因此不擅長處理 Skew Probability Distribution。而從表 4-2 可以看出整個 AlexNet 的 Symbol 分布，發現最常出現的

前十種 Symbol，其機率和約莫 30%，顯然 AlexNet 就是個 Skew Probability Distribution。因此 Arithmetic Coding 效果比 Huffman Coding 還好是正常的事。

Ranking	Ascii Code	Probability
1	188	6.3808%
2	187	5.8288%
3	59	4.8637%
4	60	4.7608%
5	183	1.8270%
6	186	1.7223%
7	58	1.5830%
8	23	1.4545%
9	73	1.2712%
10	237	1.2633%

表 4-2 AlexNet 的 Symbol 機率排名表

而從表 4-1 中 Binary Arithmetic Coding 的結果，也能注意到 Huffman Coding 對於每個 Symbol 的編碼都是整數長度這件事，是一個不可忽略的缺點。如果 Huffman Coding 只能編碼兩種 Symbol，無論兩種 Symbol 的機率分布為何，最後一定編碼成 0 和 1 兩種長度為 1Bit 的符號。也就是說，如果使用 Huffman Coding 編碼兩種 Symbol，不但沒壓縮任何空間，還要多花費 Header 的成本，只會越壓越大，而 Arithmetic Coding 對於單一 Symbol 的平均編碼長度可以是小數，所以 Binary Arithmetic Coding 還是可以壓縮檔案。

繼續觀察表 4-1 中的 FPM 與 PPM with 0-Order 的差異，我們發現不論是 Binary Arithmetic Coding 還是 256-Symbol Arithmetic Coding，其 FPM 或 0-Order PPM 的 Expected Coding Length 大致相同。其實這個結論很合理，因為 0-Order PPM 其實就是 Adaptive Arithmetic Coding。道理就如同 Huffman Coding 與 Adaptive Huffman Coding，當 Source Data 大小足夠大時，Adaptive 得到的 Expected Coding Length 基本上會收斂到差不多的長度。

再繼續觀察不同 Order 的 PPM，其 Expected Coding Length 之間的差異。我們可以注意到不論是 Binary Arithmetic Coding 還是 256-Symbol Arithmetic Coding，每當 Order 增加時，Expected Coding Length 就會減少，這個結論也很合理，因為 Order 增加，就代表條件機率增加，使得 Symbol 的分布更加傾斜。雖然說 Order 增加可以使得 Expected Coding Length 變短，但也會造成 Encode 和 Decode 時間大幅增加，所以實務上也要在壓縮時間和壓縮大小間取得平衡。

Algorithm	Total Size
256-Symbol Huffman Coding	213494662 Bytes
16Bit Huffman Coding	131245413 Bytes
24Bit Huffman Coding	124587312 Bytes
32Bit Huffman Coding	68159596 Bytes
256-Symbol Adaptive Huffman Coding	213494623 Bytes
256-Symbol AC + FPM	212674286 Bytes
256-Symbol AC + PPM Order = 0	212673778 Bytes
256-Symbol AC + PPM Order = 1	86848396 Bytes
256-Symbol AC + PPM Order = 2	69065241 Bytes
Binary AC + FPM	244213459 Bytes
Binary AC + PPM Order = 0	244213457 Bytes
Binary AC + PPM Order = 1	244162259 Bytes
Binary AC + PPM Order = 2	243810896 Bytes

表 4-3 演算法與壓縮檔 Total Size 關係

Algorithm	壓縮後前比值
256-Symbol Huffman Coding	87.3513 %
16Bit Huffman Coding	53.6990 %
24Bit Huffman Coding	50.9749 %
32Bit Huffman Coding	27.8875 %
256-Symbol Adaptive Huffman Coding	87.3513 %
256-Symbol AC + FPM	87.0156 %
256-Symbol AC + PPM Order = 0	87.0155 %
256-Symbol AC + PPM Order = 1	35.5340 %
256-Symbol AC + PPM Order = 2	28.2580 %
Binary AC + FPM	99.9199 %
Binary AC + PPM Order = 0	99.9199 %
Binary AC + PPM Order = 1	99.8990 %
Binary AC + PPM Order = 2	99.7552 %

表 4-4 演算法與壓縮後前比值關係

表 4-3 和表 4-4 可以一起看，可以發現就算將 Header 也納入考量，壓縮後前比值會隨 Symbol 的 Bits 數減少而增加，且在 Binary Arithmetic Coding 時，壓縮後仍然比壓縮前大小還小。

同樣的從表 4-3 和表 4-4 可以看出在壓縮 AlexNet 的 Model 時，若不是只看 256-Symbol Huffman Coding 和 256-Symbol AC + FPM 。在這次的模型壓縮來說，Arithmetic Coding 和 Huffman Coding 的結果其實沒差太多。雖然說 AlexNet 的 Symbol 分布傾斜，但是在 32Bit Huffman Coding 的情況下，Huffman Coding 的 CodeBook 也才 7000 多種 Symbol，CodeBook 並不是大到過於佔空間。這也造成了 32Bit Huffman Coding 和 256-Symbol AC+ PPM with Order = 2 有差不多的結果。

V. Conclusion

從這次的報告中，我除了學到如何用 Python 實作整數版本的 Arithmetic Coding，還驗證了上課提到 Arithmetic Coding 和 PPM 的好處。如果想做更進一步的探討，或許可以朝向比較 PPMA，PPM Method B，PPM Method C 的差異，和使用 Exclusion Principle 的效果。

References

[1] N. Abramson. Information Theory and Coding. McGraw-Hill, 1963.

[2] J.J. Rissanen and G.G. Langdon. Arithmetic coding. IBM Journal of Research and Development, 23(2):149–162, March 1979

[3] J.G. Cleary and I.H. Witten. Data compression using adaptive coding and partial string matching. IEEE Transactions on Communications, 32(4):396–402, 1984.

[4] A. Moffat. Implementing the PPM Data Compression Scheme. IEEE Transactions on Communications, Vol. COM-38:1917–1921, November 1990.