Adaptive Huffman Coding: Analysis and Applications

Jiaxin Li Benhao Huang Dingyao Tao

May 2022

Abstract

We start with the Bad Wine problem from the textbook and illustrate the role of Adaptive Huffman coding by raising a new scenario "Bad Wine Pipeline". Then we describe the process of implementing Adaptive Huffman coding.

In the next section, we analyze the performance of Adaptive Huffman Code in the view of a encode method. In our experiment, we test the compress rate, encode time and decode time in python. After analyze the result, we can conclude that the best interval of the size of message is $[2^7, 2^{12}]$.

Then we focus our attention on its learning velocity and its adaptability. We define a series of reasonable and intuitive metrics to measure its learning velocity and analyze its learning velocity in the different distribution. Based on the above experiment result, we propose the Adaptive Huffman Code with Buffer to improve its learning velocity. Besides, our experiment verify its feasible.

Finally, we explore the application of Adaptive Huffman coding to the CBOW hierarchical softmax method. We present a different viewpoint on the Incremental Method approach proposed in existing papers and improve it with the Adaptive Huffman Tree, which we call Partial Incremental with Adaptive Huffman(PIWA). Further experiments show that our improvements outweight the incremental method both in final accuracy and predicting time, while basically has the same training cost with it.

Keywords: Adaptive Huffman coding, Algorithm performance, Learning velocity, Hierarchical softmax, Incremental learning

1 Introduction

Our research is inspired by a problem in Cover's textbook[1] (see page 153).

Example 1 (Bad wine). *One is given six bottles of wine. It is known that precisely one bottle has gone bad (tastes terrible). From inspection of the bottles it is determined that the probability p_i that the ith bottle is bad is given by $(p_1, p_2, \ldots, p_6) = (\frac{8}{23}, \frac{6}{23}, \frac{4}{23}, \frac{2}{23}, \frac{2}{23}, \frac{1}{23})$. Tasting will determine the bad wine. You can mix some of the wines in a fresh glass and sample the mixture. You proceed, mixing and tasting, stopping when the bad bottle has been determined. What is the minimum expected number of tastings required to determine the bad wine?*

As known well, this problem can be solved using Huffman code. Here we render the corresponding Huffman tree as solution and give no more elaboration here.

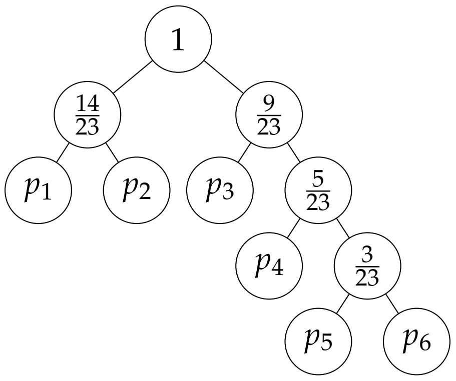


Figure 1: Huffman Tree of Example 2.

However, this easy case is what we call "still". What if the probability of a certain wine is bad could be dynamically changing through the time? What if the number of the type of wines is not fixed as 6 in example 2 and can be increasing by the time?

In order to give a more intuitive feeling, we come up with the following new application scenario, we call it "Bad Wine Pipeline".

Example 2 (Bad Wine Pipeline). *There is a wine pipeline with four product lines, labeled A, B, C, and D. The probability of bad wine being produced is unknown for each line. At the end of the pipeline, a robot checks the quality of the wine to determine if any of it is bad.*

1)You are tasked with developing a strategy that enables the robot to efficiently find bad wine with minimum average checks. The robot is able to mix some wines and check them together. Note that there is a feedback loop; when the robot discovers bad wine from a certain line, the corresponding probability will be updated. Therefore, your strategy should be able to cope with dynamic changes.

2)Your boss wants to expand the business by adding new lines, such as E, F, G, and so on. Does your strategy still work in this case?

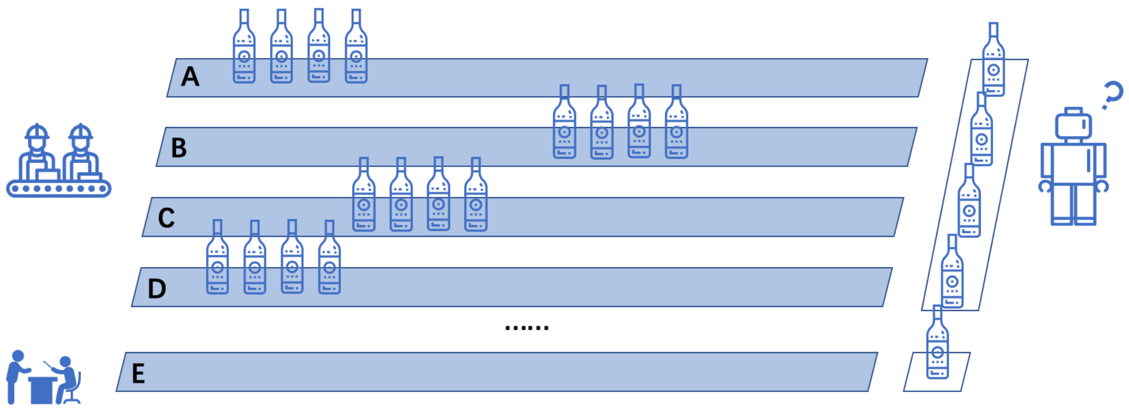


Figure 2: Bad Wine Pipeline

Naive Huffman coding won't work in this scenario. Intuitively, we have to make the coding changing in the procedure, this is what Adaptive Huffman coding (also called Dynamic Huffman coding) do.

Adaptive Huffman coding[2] permits building the code during the transmission of the symbols, where we might have no initial knowledge of source distribution. Thus, it can be used to develop one-pass[1] algorithms for file compression, which can be useful in situations where there is limited memory or processing power available. [3]

2 Adaptive Huffman Code

2.1 Basic Process

The best way to learn an algorithm is to follow it step by step. Before we demonstrate how it works, there are some definitions should be made clear.

Definition 1. *NYT node stands for 'Not Yet Transmitted'. In the building procedure of an adaptive Huffman tree, it serves as auxiliary node. In data compression and transmission procedure, it's an escape code. When a symbol not yet contained in the coding tree needs to be encoded, the code of NYT is first output, followed by the symbol's original expression specified by the sender and the receiver. When the decoder encounters a NYT, it knows that what follows is temporarily no longer a Huffman encoding, but a primitive symbol that has never appeared in the encoded data stream.*

Definition 2. *The weight of a node is the number of times it appears, which is denoted by the number in the bracket of the node.*

[1]In computing, a one-pass algorithm or single-pass algorithm is a streaming algorithm which reads its input exactly once.

Definition 3. *Block: a group of nodes who have the same weight.*

Definition 4. *Node Number: The number of a node is up to the structure of Adaptive Huffman Tree. Starting at 1, it increases from left to the right, bottom to the top.*

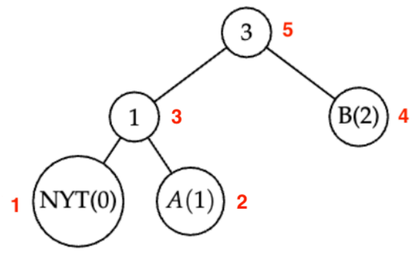


Figure 3: the number of nodes

Definition 5. *Sibling Property : In the process of constructing dynamic Hoffman coding trees, two important principles need to be followed:*

(1) The nodes with larger weight also have larger node numbers.

(2) The node number of the parent node is always larger than the node number of the child node.

Then we introduce the process of building and updating of a Adaptive Huffman Tree. Initially, frequency(or probability) of various symbols is unknown. We specify that the initial state has only one leaf node: NYT. Every time a symbol is inserted, we should keep the sibling property while updating the weight. When trying to insert a symbol, based on whether it is in the tree, we have different strategy. Particularly, If the symbol has been encountered, then we should build a subtree correspondingly.

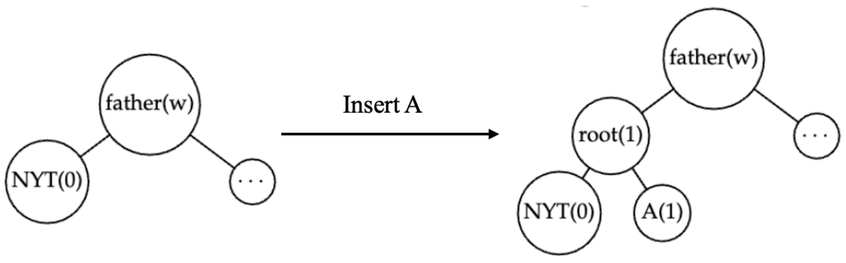


Figure 4: Insert a new symbol

The flow diagram demonstrates more details about the procedure is included in appendix 7.1.

3 Performance of Compression: Contrast

In order to research on the Adaptive Huffman code better, we analyze its performance in many aspect in this section.

3.1 The performance of Adaptive Huffman vs the performance of adaptive arithmetic

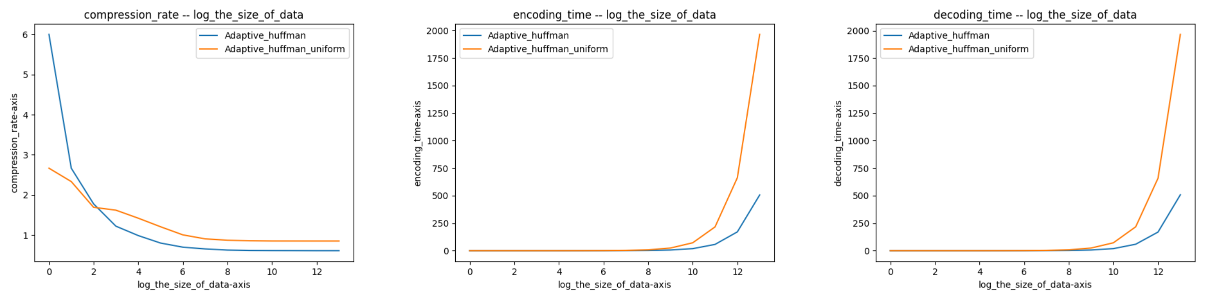
When considering a data compression encoding method, several aspects should be considered:

Definition 6. *Compression rate: This determines the minimum size of the data we can achieve from the compression encoding method.*

Definition 7. *Encoding time: This refers to the time cost when encoding data.*

Definition 8. *Decoding time: This refers to the time cost when decoding the code.*

Here's the result:



compression rate   encoding time   decoding time

Figure 5: Visual comparisons of the Adaptive Huffman compression encoding models with adaptive arithmetic compression encoding model.

3.2 Result Analyze and Conclusion

From the figure fig4, we can conclude that :

1. the compression rate is decrease with the increase of the size of message and finally becomes stable.
2. the encoding time and decoding time "explode" after the size of message exceed the threshold. s

3. After comprehensive analyzing the result, the Adaptive Huffman Code will get the best performance in our experiment environment when the size of message is $[2^7, 2^{12}]$.

4 Explore:"learning" convergence velocity

After analyze the conventional performance of Adaptive Huffman Code in the perspective of a code method, we want to explore its special performance from its adaptability.

Learning from two algorithm , it is clear that Adaptive Huffman Code updates its Huffman Code Tree in the process of coding, in order to adapt to the change of probability distribution. It is obviously that once the distribution changed or in the beginning of coding(Figure 6), the Huffman Code Tree is not optimal. If the Huffman Code Tree is able to "learn" the optimal Huffman Code Tree rapidly, the length of encode will be shorter and the compress ratio of this encode method will be better.

Therefore, it is significant that explore how to measure the "learning" velocity and what to affect the "learning" velocity, how to improve the "learning" velocity.In the following section, we will explore the "learning" velocity in this three aspect.

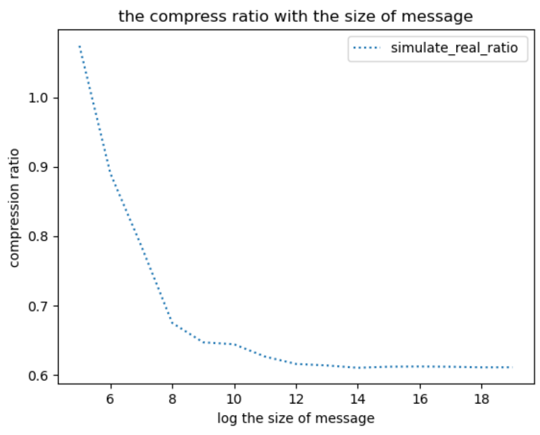


Figure 6: The compression ratio with the size of message

4.1 The Metrics of the "learning" velocity

Before we introduce the metrics of the "learning" velocity, let us analyze figure 6 formally: we use the letter probability distribution of to generate the different length of text. This figure demonstrate the relationship between compression ratio and the length of the text to be encoded.

In this figure, we find that the compression ratio is decreasing with the increasing of length of message and the decrease velocity become less and less.Therefore, we introduce the definition of the convergence state.

Definition 9. *the convergence state: when the length of message is doubled, the decrease of compression ratio is less than* ***0.01*** *continuously.we define this state as* ***the convergence state****:*

Then, we define a series of metrics to measure the convergence velocity (in other word, the "learning" velocity).

Definition 10. *Warm-up length:*

$$L_{warm} := log(L_{minc}) \tag{1}$$

where L_{minc} :the minimum length of the message in the convergence state.

Definition 11. *Launch efficiency: the compression ratio corresponding to the length of the input message is short (2^5 letters)* [2].

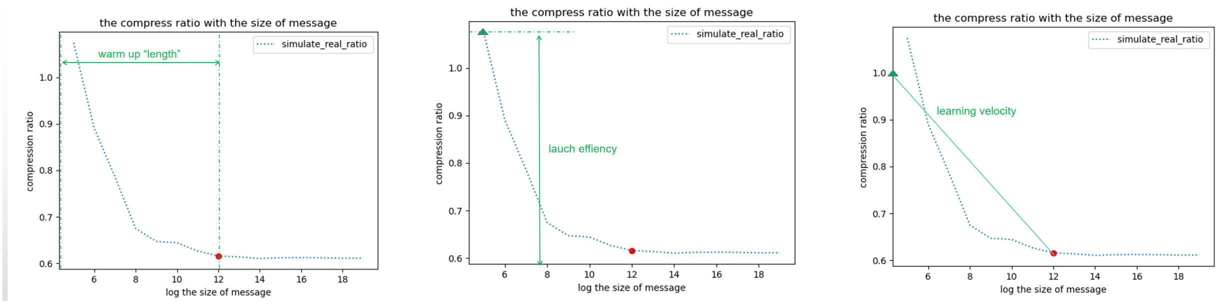
Definition 12. *Valid compression ratio*

$$R_v := 1 - R_c \tag{2}$$

where R_c is the compression ratio in the convergence state

Definition 13. *Learning velocity:*

$$v_{learning} := \frac{R_v}{L_{warm}} \tag{3}$$



Warm-up length

Launch efficiency

Learning velocity

Figure 7: Visual comparisons of original models.

[2]Based on the fact that the too short message is unnecessary to use the Adaptive Huffman Code to code whose the probability distribution of words we can get easily, we define the shortest message using Adaptive Huffman Code to code is consist of 2^5 letters. In other words, we can easily use the first 2^5 letter in the message to get the initial tree. It is one of the reason why the shortest length in the Figure 6 is 2^5

4.2 The Affect Factor: The probability distribution

It is universally acknowledged that the probability distribution make a big difference in the average length of the code in Native Huffman Code. In this section, Aim to explore the influence of probability distribution for the learning velocity, we using the following probability distribution to measure the learning velocity:

- the continuous uniform distribution
- the discontinuous uniform distribution
- the normal distribution ($\sigma = 26$)
- the simulation distribution (the probability distribution in <>)

4.2.1 Experiment

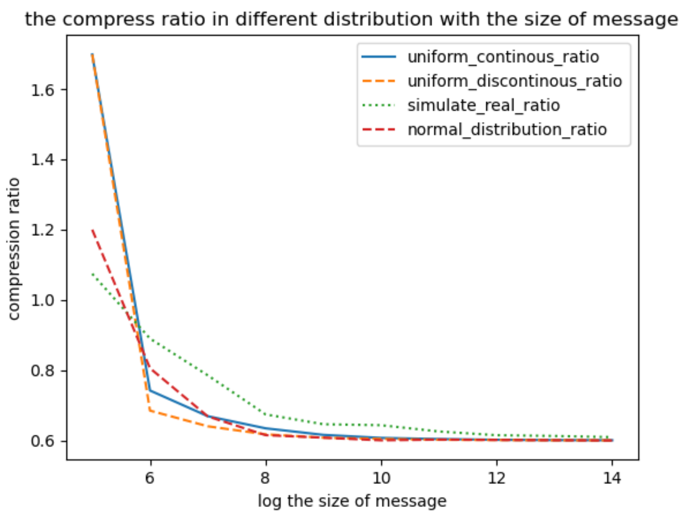


Figure 8: The compression ratio with the size of message in different distribution

Table 1: The metrics of different probability distribution

distribution	$L_{warm}/(\text{byte})$	$R_v/1$	$E_{launch}/1$	$v_{learning}/log(byte)^{-1}$
continuous uniform	2^9	0.383	1.698	0.0425
discontinuous uniform	2^{18}	0.381	1.698	0.0477
simulation	2^{12}	0.384	1.074	0.0320
normal	2^8	0.384	1.2	0.0480

4.2.2 Analyze

Table 2 and Figure 9 demonstrate:

1. the L_{warm} of normal distribution is shortest
2. The continuity of letter make a big difference on the L_{warm}
3. The distribution does not make a big impact on the R_v
4. Compare to the other distribution, the simulation distribution has the least learning velocity
5. the simulation distribution has the best E_{launch}. The uniform distribution will make the E_{launch} very bad.

In summary, the distribution probability will affect the learning velocity: the continuity will lengthen the L_{warm} .Following a certain pattern range, the message will be learned with Adaptive Huffman Code more easily.

4.3 Explore: Buffer sample

Base on the fact that if we can warm up the machine before using it, the performance of it will be better. Therefore, we make a buffer for the Adaptive Huffman Code. When we encode the message, we put the letter into the buffer firstly.Using the letter in buffer to update our Huffman Tree, only when the buffer is full or at the end of the message, we encode and output the letter in the buffer.

4.3.1 Method

In this section, we define our method formally:
In this method, although the buffer can warm up the Huffman Tree, we should deliver the statistical frequency table to the decoder in order to update the Huffman Tree at the same time which make it possible for the receiver to decode the code correctly.Therefore, we introduce the definition of the **warm up cost** to measure the additional space consumption.

Definition 14. ***Warm Up Cost***

$$C_{warm} = log|M| \times log|Size_{buffer}| \times \frac{L_{message}}{size_{buffer}} \tag{4}$$

Algorithm 1 Algorithm of Apative with Buffer

Input: message ,buffersize
Output: deliver code to the decoder

```
1: while message.remain.size > 0 do
2:     while Buffer.size <= BufferMaxSize and message.remain.size > 0 do
3:         Buffer.Push( message.remain.pop())
4:         Get the Statistical frequency table of the elements in buffer .
5:         Deliver this table to the receiver.
6:         Using this statistical frequency table to update the Huffman Tree in decoder and encoder.
7:         Using the updated Huffman Tree to encode the elements in buffer. And deliver the code to the decoder.
8:     end while
9: end while
```

where ｜M｜ is the number of categories of the element in message.

Definition 15. *New Compression Ratio*

$$R_c' = \frac{L_{code} + c_{warm}}{L_{message}} \tag{5}$$

4.3.2 Experiment and Result

We equip the Huffman Code Tree with buffer whose size = 64. Compare with the Huffman Code without buffer, we use it to encode different length of message generate from the simulated possibility distribution and contrast their compression ratio and other metrics about the convergence velocity.The result is as follow:

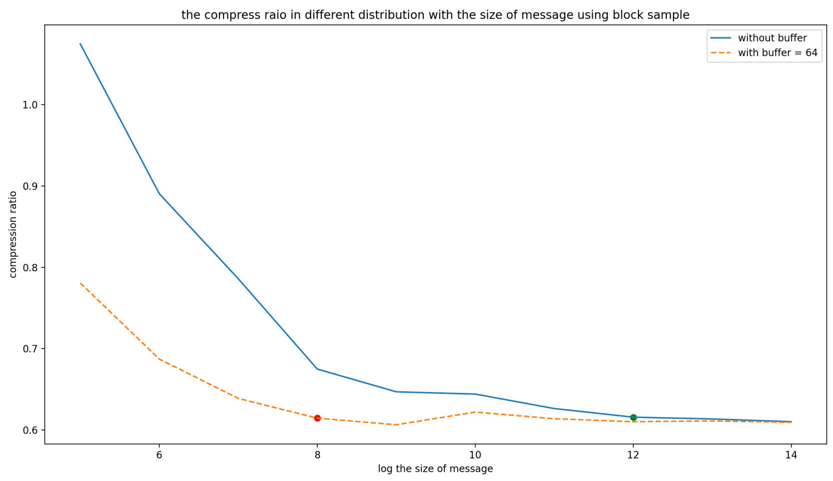


Figure 9: The compression ratio with the size of message

Table 2: The metrics of different probability distribution

distribution	L_{warm}/(byte)	$R_v/1$	$E_{launch}/1$	$v_{learning}/log(byte)^{-1}$
with buffer	2^8	0.387	**0.78**	**0.0483**
without buffer	2^{12}	0.384	1.074	0.0320

4.3.3 Analyze

Compare with the Adaptive Huffman Code without buffer, the Adaptive Huffman Code with buffer has the smaller L_{warm}, has the better E_{launch}, has higher learning velocity.Besides, their R_v have similar resultant values, which is expected to be worse in the Adaptive Huffman code[3].

In summary, we conclude that the Adaptive Huffman Code with buffer make a progress in the learning velocity in our simulation case.

5 Partial Incremental: Application of ADA in CBOW Model

5.1 Backgound

CBOW (continuous bag of words)[4] is an frequently used neural network model for word vector generation. It has multiple advantages over traditional matrix factory-based methods, while being short in its high requirement for storage and computation resources. To overcome this shortcoming, the authors of CBOW later applied hierarchical softmax[5] method to modify the model. Instead of evaluating W output nodes in the neural network to obtain the probability distribution, it is needed to evaluate only about $log2(W)$ nodes.

For clarity, we first illustrate the application of naive huffman tree behind it.

5.2 Biased Walk on Huffman Tree

Given a huffman tree, whose leaf nodes each representing a certain word w, the procedure we get the target word from input x_w can be deemed as a procedure of random walk on it.

Definition 16. *l^w: the number of nodes in p^w*

[3]The fact that this metrics is better is beyond our expectation. We make guess that the reason is the definition of R_v in this paper is not very exactly. Maybe if we define threshold, it will be reasonable in this comparison

Definition 17. *p^w: path from the root node to the leaf node corresponding to w*

Definition 18. *$p_1^w, p_2^w, \cdots, p_{l^w}^w$: the l^w th node in p^w. Here p_1^w represents the root node, $p_{l^w}^w$ represents the node of word w.*

Definition 19. *$d_2^w, d_3^w, \cdots, d_{l^w}^w \in \{0,1\}$: the huffman code of word w. d_j^w represents the code represented by the j^{th} node on p^w, where root doesn't stand for a code.*

Definition 20. *$\theta_1^w, \theta_2^w, \cdots, \theta_{l^w-1}^w \in \mathbb{R}^m$: parameter vector corresponding to non-leaf nodes in the path p^w. **Note that** the theta of a node in the tree won't varies between different word, which means a node can only have one θ value. Here the w in θ_w only tell that its on path p^w.*

The hierarchical softmax works as follows: when inputting a value X_w calculated from word w, we starting biased walking. When at node we stipulate that we have $\sigma(\theta_i^w X_w)$ chance of going right, and corresponding $1-\sigma(\theta_i^w X_w)$ chance of going left. Here $\sigma\left(\mathbf{x}_w^\top \theta\right) = \frac{1}{1+e^{-\mathbf{x}_w^\top \theta}}$. We keep walking following probability until we reach a leaf node, and output the corresponding word of the huffman code.

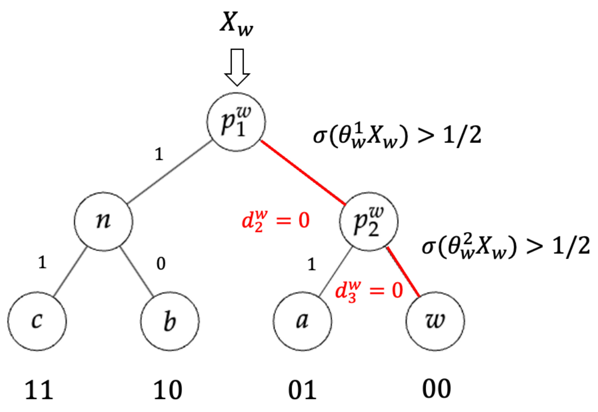


Figure 10: Example of biased walk on huffman tree

So given a word w and its huffman code, our goal is to find the proper θ_i^w to maximize the probability that we walk to the leaf node of w. Therefore, the objective function could be formulated as:

$$\prod_{j=2}^{l^w} p\left(d_j^w \mid \mathbf{x}_w, \theta_{j-1}^w\right) = \prod_{j=2}^{l^w} \left\{ \left[\sigma\left(\mathbf{x}_w^\top \theta_{j-1}^w\right)\right]^{1-d_j^w} \cdot \left[1-\sigma\left(\mathbf{x}_w^\top \theta_{j-1}^w\right)\right]^{d_j^w} \right\}, \tag{6}$$

which should be maximized after taking the logarithm during our training.

Proposition 1. *The predicting time of the model is positively correlated with the height of the tree, because larger the height, longer the path we should walk to get to the leaf node.*

5.3 A Further Look into Incremental Learning

Although hierarchical softmax could save a lot computation resource, it's still not enough. For a certain dataset, the corresponding huffman tree is unique, which means every time we have new data added into dataset, we have to retrain the model with whole When the dataset becomes too large, the cost of building a huffman tree can not be ignored. Under such circumstance, if we have already trained our model on a dataset, its expensive to retrain the model on the whole updated dataset.

5.3.1 Previous Works

There are several recent works[6] [7] putting forward a method called "Incremental Huffman Tree" to solve the problem. The basic idea of incremental huffman tree is simple and clear as demonstrated in the following pesudocode:

Algorithm 2 Incremental Huffman Tree

```
Input: previous Huffman Tree $T_p$ , new dataset
Output: updated "Huffman Tree" $T_u$
1: for word in new dataset and not in previous Huffman Tree do
2:     add word into new word set
3: end for
4: build a new Huffman Tree $T_n$ on new word set
5: Merge: find the shortest path of $T_p$ and make its leaf node $n_s$ the root of $T_n$
6: update the Huffman Code of leaf nodes in $T_n$ by adding the code of $n_s$ as prefix
```

5.3.2 Problems We Found

Compared with reconstructing a new Huffman Tree, this method apparently saves a lot computation resources. **However, we thought there are some problems of such method and make our own analysis.**

Problem 1. *The accuracy of the model will decrease with this training method.*

if we find the shortest path of T_p and make its leaf node n_s the root of T_n, then this node just disappear, since the walk on the tree won't stop until we reach the leaf node, which means the internal nodes won't be deemed as a word by the model! Moreover, since this node is on the shortest path of original **A possible solution** could be add the node n_s to the new word set then reconstruct the new tree, but authors of these two papers did not elaborate on this detail.

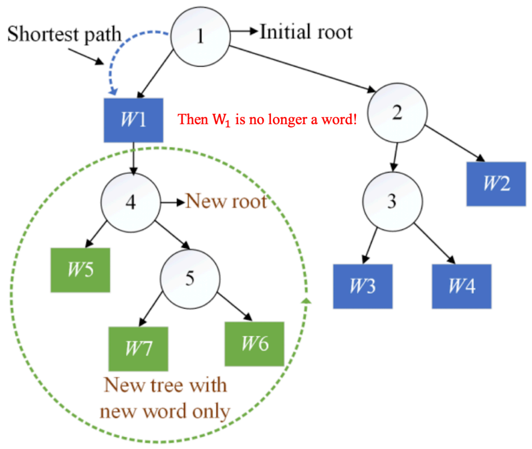


Figure 11: From M. Nilufar and A. Abhari's paper[6], the red text explains our query

Problem 2. *This method can't really implement incremental learning.*

When we refer to Incremental Learning, we usually hopes not to retrain on the whole dataset again after updating. However, this incremental Huffman Tree method can not actually achieve this effect. As we have pointed out in Definition 20, a certain node has only one parameter vector θ, if we just training the model on new words without the old words, then the parameter learned for old words will be ruined. **Considering that both of the papers [6] [7] give no open source code and are lack of results on the accuracy of the model trained, we carry out the experiment on our own.**

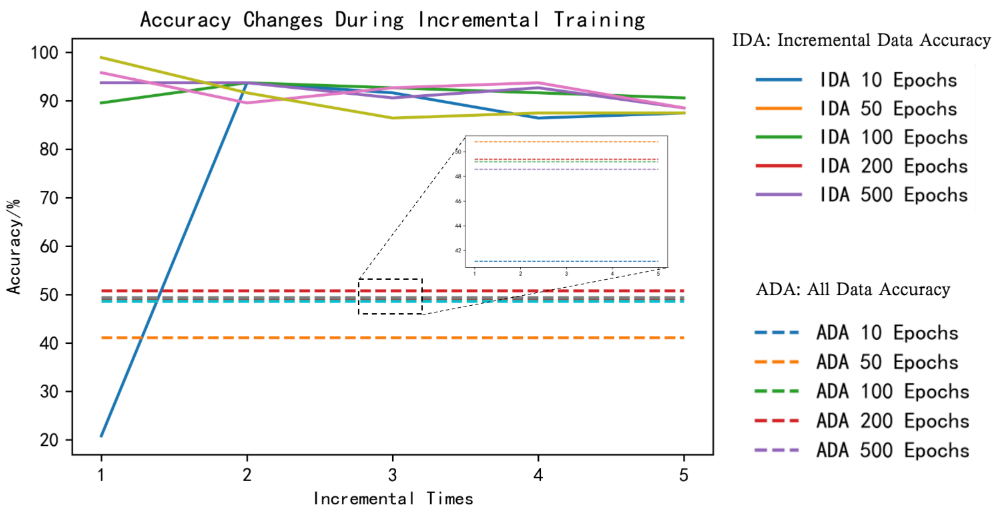


Figure 12: The accuracy changes during "Incremental Learning"

Here Incremental Data Accuracy is validated on each incremental dataset, and All Data

Accuracy refers to the accuracy on the whole dataset after finishing all training. We carry out five experiments with epochs=10,50,100,200.

From figure (12) we observe that as training epoch increases, IDA basically falls in roughly the same range, except for when epoch=10, which is obviously undertraining (at incremental time 1). While ADA first increases and then drops, which means Incremental Learning by this method has severe drawbacks. More experiments results is included in appendix 7.4.

Problem 3. *Incremental Huffman Tree's code length could be very bad, as data set becoming very large, which also means Incremental Huffman Tree has large height.*

Intuitively, as merge times increases, this incremental huffman tree may become large in height and finally have very large code length. We try to compare it with the optimal huffman tree, given binary alphabet = $\{0, 1\}$.

Theorem. *Huffman Code has minimum average code length L^* for binary alphabet.*

Proposition 2. *The Incremental Method could make code length as bad as nL^*, here $n = \log w + 1$, w is the number of dictinct words in the dataset before updating with new words.*

Assume there are 2^{n-1} distinct words in the previous data with weight 1, and we add a new word with weight m into the dataset, where $m >> 2^{n-1}$. The optimal case with Huffman Tree will produce a code with length $n2^{n-1} + m$, while the incremental method gives us a code with length $2^{n-1} + nm$.

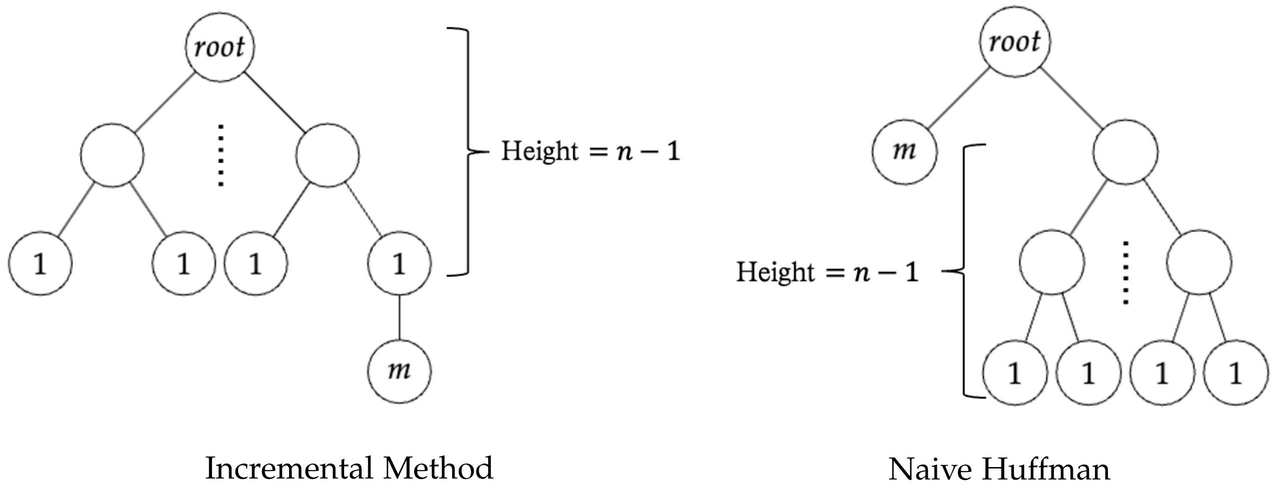


Figure 13: Comparison between the code length of Huffman Tree and Incremental Huffman

Then the code length rate between two methods is :

$$\alpha = \frac{2^{n-1}(n-1) + nm}{2^{n-1}n + m} \longrightarrow n, \text{with } m \to \infty, \tag{7}$$

which means it can consumes large storage to store the word code for some particular dateset.

Therefore, we could claim that this training method cannot serve as Incremental Learning method, which means it still needs to retrain on the whole new dataset to obtain normal accuracy. However, as we have pointed out in Problem 1 and 3, its accuracy could be ruined and its code length could be extremely large, which harms training equality. Thus, combined with Adaptive Huffman Tree we have explored above, we put forward a partial incremental method.

5.4 Our Refinement: Partial Incremental With Adaptive Huffman Tree

Because of the special structure of hierarchical softmax, we could hardly realize Incremental Learning only on the new words added, as we have exlained in problem **??**. What we could do is to optimize the reconstruction procedure of hierarchical softmax tree. We could simply use Adaptive Huffman Tree to replace it,and ensure the minimum length of code while saving the time cost of reconstructing a new Huffman Tree. Our algorithm for is designed as follows:

Algorithm 3 PIWA:Partial Incremental With Adaptive Huffman

```
Input: previous Huffman Tree $T_p$ , new dataset
Output: updated Huffman Tree $T_u$
1: for word in new dataset do
2:     use Adaptive Huffman Tree algorithm to update the tree
3: end for
4: $dataset \leftarrow old\ dataset \cup new\ dataset$
5: Retrain the model on dataset
```

Then we carry out some experiment testing its accuracy, training time cost and predicting time cost compared to Incremental Huffman methods. The dataset we use is included in appendix. Limited by our time and computer performance, we set incremental times as 8 and batch size as 100 words, which means totally we have 800 words.

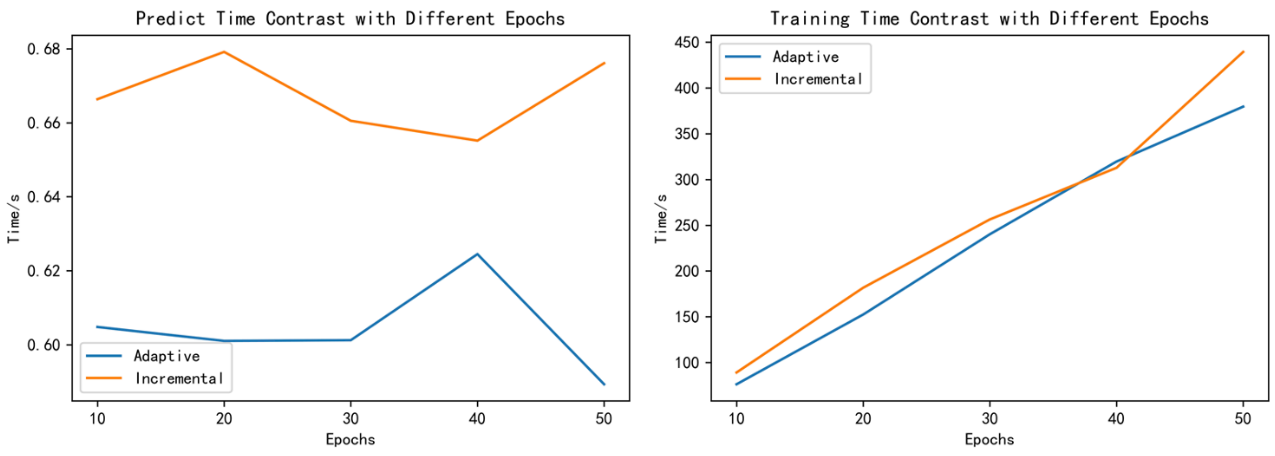


Figure 14: Contrast on Training and Predicting Time with Different Epochs

From figure 14, we find that the Adaptive Huffman always use less predicting time. As we argued in proposition 1, predicting time is mainly related to the average tree height, thus, we conclude that Incremental Tree's height is larger than Adaptive Tree, which prove the existence of Problem 3.

As for training time, the training time of the Adaptive method increases more steadily with the increase of the epoch, while the Incremental method has a change in slope, which probably means that the Incremental method will get worse with the increase of the epoch.

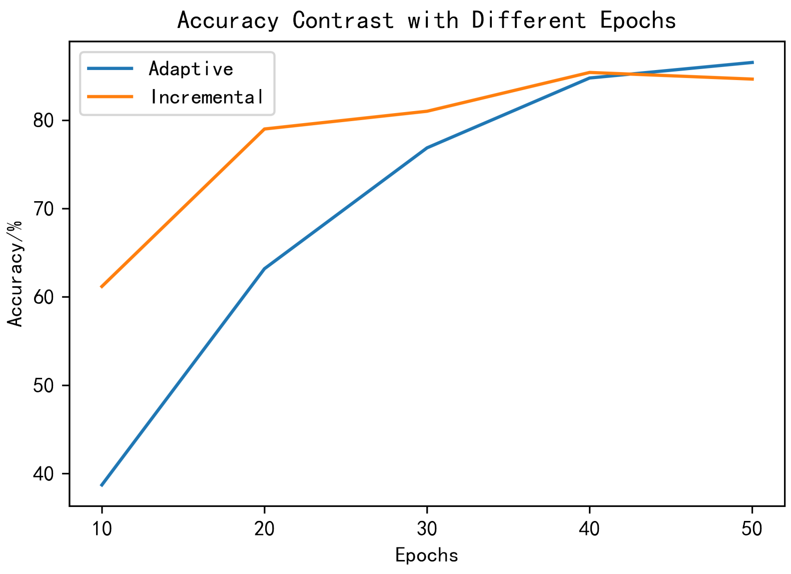


Figure 15: Contrast on Accuracy with Different Epochs

In terms of accuracy, it can be seen from the figure 15 that with the increase of epoch, the accuracy of Incremental method increases gently. When epoch=50, it is overtaken by Adaptive method. This points to the Incremental method we argued in Problem 1 that it undermines accuracy by lowering the upper bounds for accuracy because words disappear from its model.

Therefore, we could conclude that PIWA algorithm is more efficient than Incremental method, which serves as an improvement.

6 Future Work

6.1 Adaptive Huffman Code With Buffer

- We just implemented the process of encode and analyzed its performance. The process of decoder can be implemented in the future to make the Adaptive Huffman Code With Buffer complete.
- The metrics can be defined more appropriate for the experiment for the Adaptive Huffman Code With Buffer. If time permit, we can explore deeply in this aspect.

6.2 Modification of Partial Incremental Algorithm

- We haven't combine the research on improving the performance of Adaptive Huffman Tree with application scenario.
- Partial Incremental still has still a long way from Incremental Training. To truly implement Incremental Training, we may have to modify the whole structure of the hierarchical softmax model.
- Our implementation of Adaptive Huffman Tree should be further improved with heap to cut down time complexity more.

7 Conclusion

In this paper, introduced by "the Bad Wine Pipeline", we described the basic process of a high-level encode method "Adaptive Huffman Code" which permit the probability distributions of words change.

Then in order to have a deeper understanding on this method, we made some experiment using this method. We analyze its performance about the code in many aspect, including compression rate, encoding time and decoding time. As the experiment showed, the encode time and the decode time will increase rapidly once the size of message is more than threshold 2^{12}. Combined with the compression rate, we concluded that the best interval of the size of message is $[2^7, 2^{12}]$.

After analyzing its performance about the code, we explored its learning velocity given its adaptability. We defined a series of reasonable metrics to measure its earning velocity. Using these metrics, we explore how the probability distribution affect its learning velocity.Finally, in order to improve its learning velocity, we proposed the Adaptive Huffman Code with Buffer Method and our experiment verified its feasibility.

Finally, we attempt to apply Adaptive Huffman coding in a hierarchical softmax approach of the CBOW model. We first analyse the possible problems with the Incremental Huffman Tree proposed by existing works, then we propose the PIWA algorithm incorporating Adaptive Huffman coding. Through experimental verification we find that our method outperforms the Incremental Huffman algorithm, with higher accuracy, less predicting time and almost the same training time.

References

[1] T. M. Cover, *Elements of information theory*. John Wiley & Sons, 1999.

[2] D. E. Knuth, "Dynamic huffman coding," *Journal of Algorithms*, vol. 6, no. 2, pp. 163–180, 1985. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0196677485900367

[3] Z. Li, M. Drew, and J. Liu, *Fundamentals of Multimedia*, ser. Texts in Computer Science. Springer International Publishing, 2014. [Online]. Available: https://books.google.com.sg/books?id=R6vBBAAAQBAJ

[4] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[5] F. Morin and Y. Bengio, "Hierarchical probabilistic neural network language model," in *International workshop on artificial intelligence and statistics*. PMLR, 2005, pp. 246–252.

[6] M. Nilufar and A. Abhari, "Incremental text clustering algorithm for cloud-based data management in scientific research papers," in *2022 Annual Modeling and Simulation Conference (ANNSIM)*, 2022, pp. 778–789.

[7] L. Tian, X. Wen, Z. Song *et al.*, "An online word vector generation method based on incremental huffman tree merging," *Tehnički vjesnik*, vol. 28, no. 1, pp. 52–57, 2021.

Appendix

7.1 The Flow Diagram of Adaptive Huffman Tree Algorithm

The detailed procedure of Adaptive Huffman Tree Algorithm is as follows:

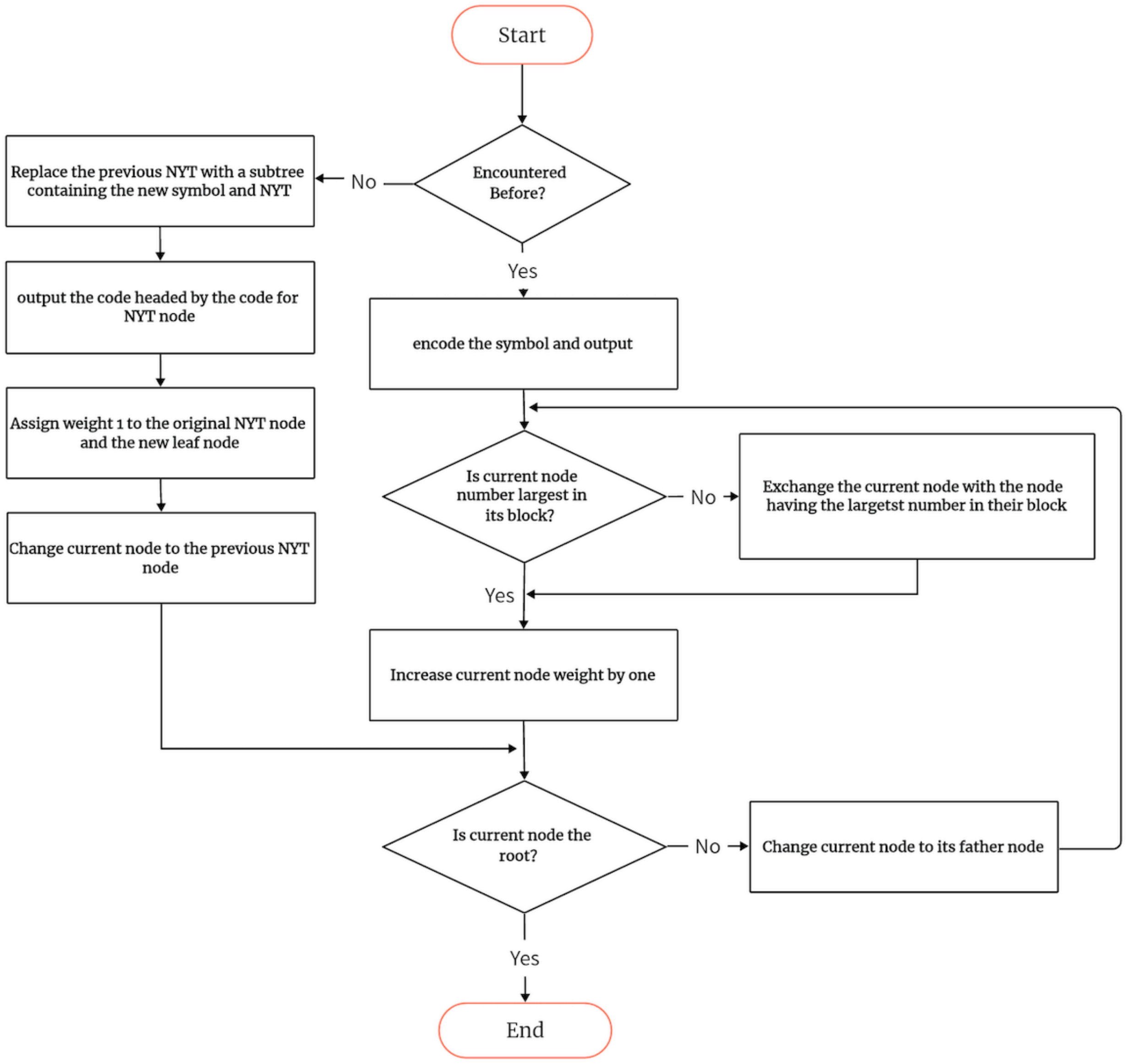

Start
Encountered Before?
No
Replace the previous NYT with a subtree containing the new symbol and NYT
output the code headed by the code for NYT node
Assign weight 1 to the original NYT node and the new leaf node
Change current node to the previous NYT node
Yes
encode the symbol and output
Is current node number largest in its block?
No
Exchange the current node with the node having the largetst number in their block
Yes
Increase current node weight by one
Is current node the root?
No
Change current node to its father node
Yes
End

Figure 16: building and updating Adaptive Huffman Tree

7.2 Code Repository

All the codes are available in https://github.com/huskydoge/Exploration-on-Adaptive-Huffman/tree/main

7.3 Dataset

The dataset we used is from ...., and we have included in
https://github.com/huskydoge/Exploration-on-Adaptive-Huffman.

7.4 Additional Experiment On Incremental Huffman Tree

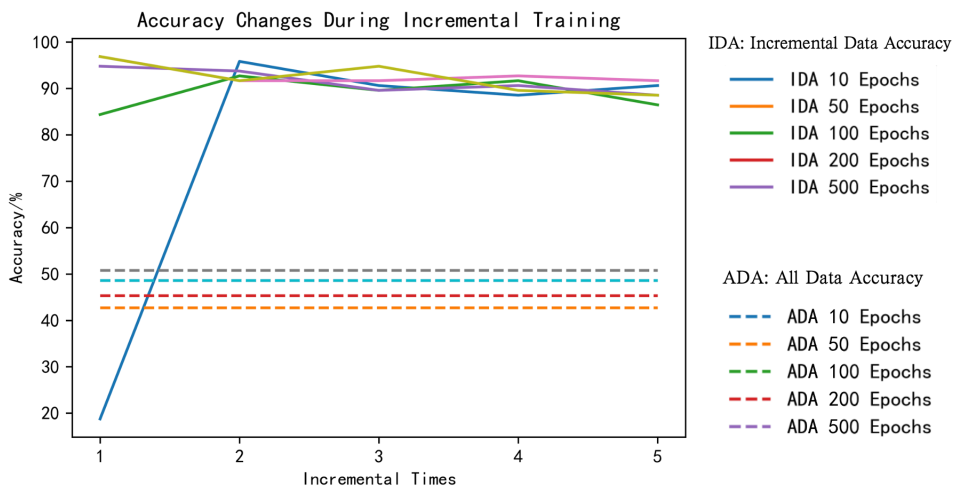


Figure 17: Repeated Experiment On Incremental Huffman Tree

As we could see above, the trends reflected from the chart are consistent with those we analyzed in the previous chart.

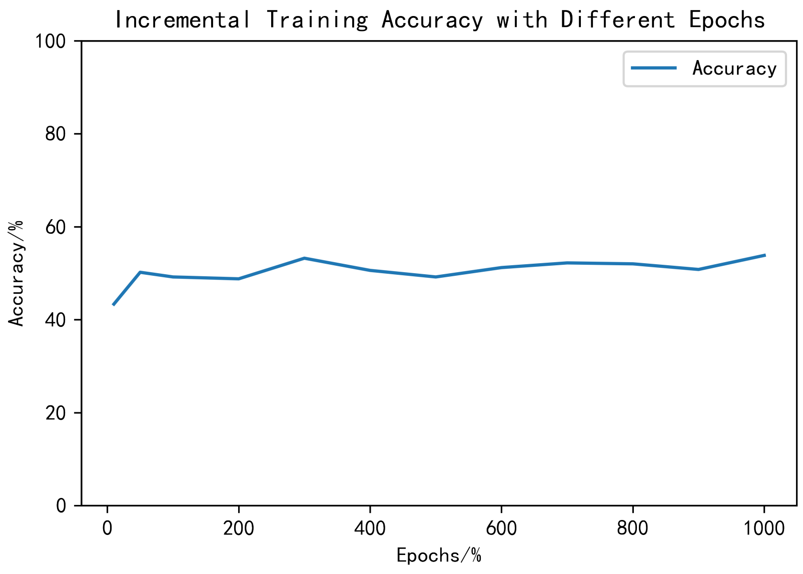


Figure 18: Incremental Training Accuracy with Different Epochs

From the chart above, we could see that the accuracy basically fluctuates around 50% while increasing the number of epochs.

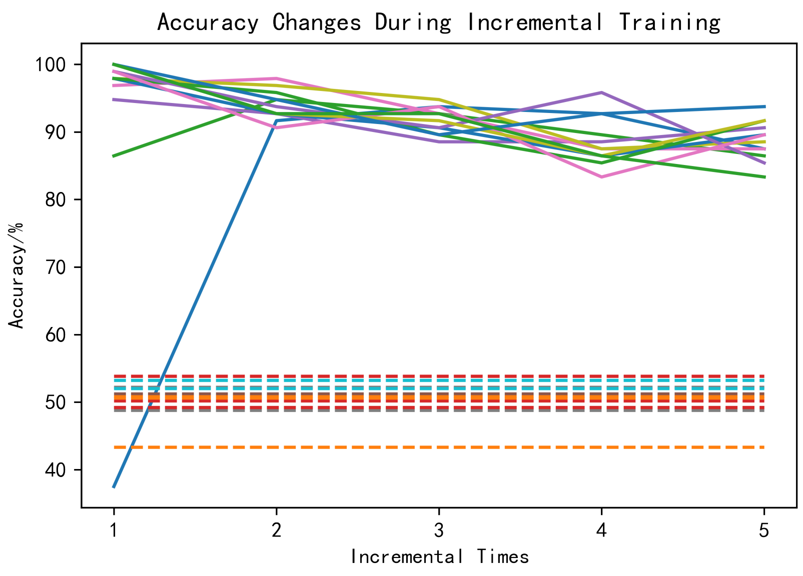


Figure 19: Condtions with more epochs

If we keep adding the epochs, we can observe that the accuracy rate on each incremental dataset has trend of decreasing.

Here we choose epochs = 10,50,100,200,300,400,500,600,700,800,900,1000.