HuffmanEmbed: Using Huffman Coding for Embedding Table Compression in Deep Learning Recommendation Models

Chaoyi Jiang*
Abdulla Alshabanah*
chaoyij@usc.edu
aalshaba@usc.edu
University of Southern California
Los Angeles, California, USA

Hossein Entezari Zarch
University of Southern California
Los Angeles, USA
entezari@usc.edu

Keshav Balasubramanian
University of Southern California
Los Angeles, USA
keshavba@usc.edu

Murali Annavaram
University of Southern California
Los Angeles, USA
annavara@usc.edu

Abstract

Deep Learning Recommendation Models (DLRMs) have become widely popular for click-through rate (CTR) prediction tasks. These models often rely on embedding tables to enhance their predictive performance. Each row in these tables represents a trainable weight vector associated with a specific feature instance (embedding index) of a categorical feature. The embedding table sizes have grown into Terabytes over time as model accuracy improves with larger table sizes. Large models put significant burden on GPU memory and compute resources. To solve this burden prior work employed embedding table compression schemes, but at the cost of reduced model accuracy.

This paper introduces **HuffmanEmbed**, a novel embedding table compression framework that improves accuracy at a given compression ratio. This work is based on the observation that not every categorical feature instance should be given the same learnable parameter width. Categorical feature instances have highly skewed access counts in any training dataset. HuffmanEmbed encodes embedding indices into varying-length unique codes based on their access frequencies. The embedding tables themselves are restructured to exploit the varying-length codes enabling significant model compression while preserving crucial information for accurate prediction.

1 HuffmanEmbed

1.1 HuffmanEmbed Overview and Applicability

We design HuffmanEmbed, an efficient embedding table compression framework as shown in Figure 1. There are multiple steps shown in the figure and we describe these steps in this section. The key idea of HuffmanEmbed is to generate unique feature representation leveraging statistical information like access frequencies of feature instances by using the Huffman Coding algorithm, and allocate different resources to those instances while reducing the overall memory footprint.

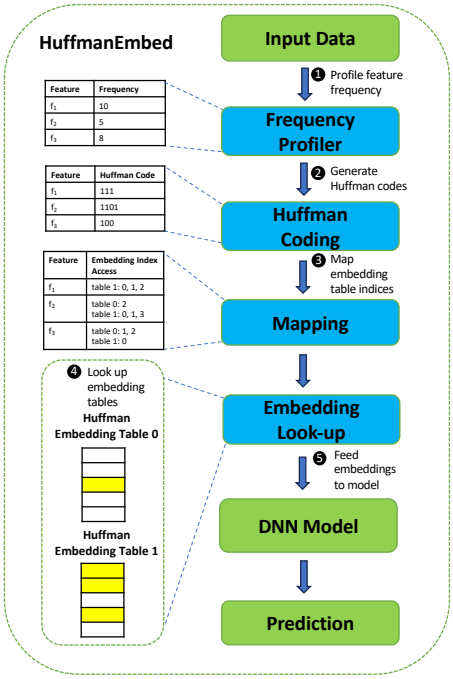

Figure 1. Overview of HuffmanEmbed.

Though the Huffman Coding algorithm has been widely applied to achieve optimal lossless compression in areas ranging from text and image compression to communication protocols and archival storage formats, we adapt it in novel ways for embedding table compression in DLRMs as it guarantees one-to-one mapping from each embedding index to its corresponding code for each feature instance. That is no two feature instances will share exactly the same Huffman code and exactly the same dense representation, which makes HuffmanEmbed distinct from other hash-based compression methods. In addition, Huffman codes are generated based

*Both authors contributed equally to this research.

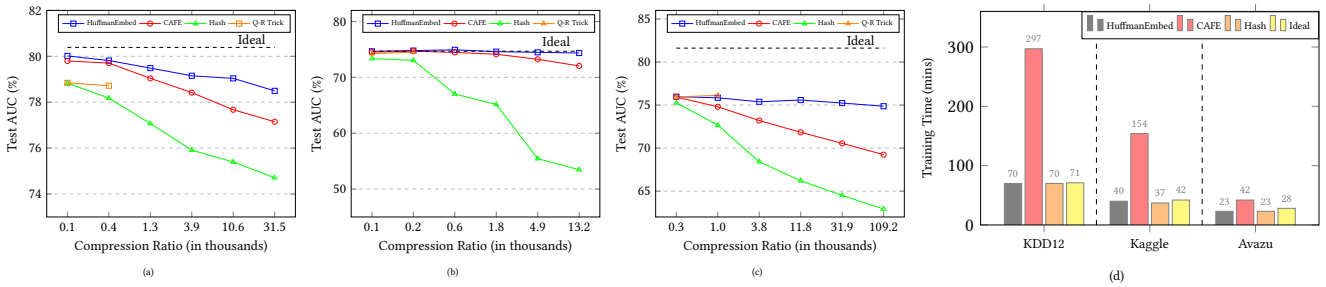

Figure 2. (a) AUC Kaggle. (b) AUC Avazu. (c) AUC KDD12. (d) Server GPU Training Time

on access frequency, which aligns with the motivation of leveraging frequency information to compress embedding tables.

1.2 Reorganizing Embedding Tables for Huffman Encoded Index

The prior section describes how embedding table indices are transformed into Huffman encoded indices. The next step (labeled steps 3 and 4 in Figure 1) is to reorganize the embedding tables to exploit the Huffman encoded index format and remap the original embedding indices. For a N-ary Huffman encoded index, our approach creates N new Huffman embedding tables. Each table has as many entries as the maximum code length L_{max}. Thus, the N Huffman embedding tables become the new model parameters to train. It is this innovative transformation of the original embedding tables into N Huffman embedding tables that creates an opportunity to achieve high compression ratios. Once the embedding tables are reorganized, we need to map the N-ary Huffman encoded index into these tables.

1.3 Index Value Mapping Algorithm

The next step (labeled step 5 in Figure 1) is to look up the embedding table using the Huffman encoded index. We create a novel lookup algorithm called the *Index Value Mapping* algorithm. In the index value mapping algorithm, we combine both the index and the value for each symbol in the Huffman codes to generate unique representations of feature instances. Specifically, under the N-ary Huffman Coding, we have N Huffman embedding tables $HE_i \in \mathbb{R}^{L_{max} \times D}$ where $1 \leq i \leq N$ and L_{max} is the maximum code length of the Huffman codes. Given a Huffman code h_i of an embedding index i, we will access $len(h_i)$ embedding rows in the Huffman embedding tables. Then, the Huffman embedding of the embedding index i is represented as: $e_i = \text{pooling}\Big(\{HE_{h_i}[j,:] \mid j \in \{0, 1, \ldots, \text{len}(h_i) - 1\}\}\Big)$, where $h_i[j]$ is the j-th bit of h_i, $HE_{h_i[j]}$ is HE_k if $h_i[j] = k$. $HE_{h_i[j]}[j,:]$ is the j-th row of table $HE_{h_i[j]}$; $\text{pooling}(HE_{h_i[0]}[0,:], HE_{h_i[1]}[1,:], \ldots)$, is a pooling function to process $len(h_i)$ embedding rows, and can be any of the well-known pooling functions, such as sum, mean or max of the embedding look-ups. To summarize, we use the values of the Huffman code to decide which table we are going to access, and use the indices of the Huffman code to select which row in the Huffman embedding table to access.

2 Overall Performance

We compare HuffmanEmbed with the Hashing Trick (Hash) [3], Q-R Trick [2], CAFE [4] and uncompressed embedding tables (labeled as Ideal) using the Area Under the Curve (AUC). We used Criteo Kaggle, Avazu, and KDD12 datasets in our experiments. It is worth nothing that in industry, even a tiny (e.g., 0.1%) reduction in AUC degradation can help recover revenue that would otherwise be lost [1].

We demonstrate the overall performance and training time in Figure 2 where HuffmanEmbed consistently achieves the lowest AUC degradation for all compression ratios without sacrificing training time.

3 Conclusion

This paper presents HuffmanEmbed, an innovative embedding compression framework designed to efficiently handle a wide range of compression ratios. It uses a lightweight pipeline with Huffman encoding and mapping modules, generating unique codes for feature instances based on frequency. The mapping algorithm then enables efficient access to the compressed embedding tables.

References

[1] Keshav Balasubramanian, Abdulla Alshabanah, Joshua D Choe, and Murali Annavaram. cdlrm: Look ahead caching for scalable training of recommendation models. In *Proceedings of the 15th ACM Conference on Recommender Systems*, RecSys '21, page 263–272, New York, NY, USA, 2021. Association for Computing Machinery.

[2] Hao-Jun Michael Shi, Dheevatsa Mudigere, Maxim Naumov, and Jiyan Yang. Compositional embeddings using complementary partitions for memory-efficient recommendation systems. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 165–175, 2020.

[3] Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. Feature hashing for large scale multitask learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 1113–1120, 2009.

[4] Hailin Zhang, Zirui Liu, Boxuan Chen, Yikai Zhao, Tong Zhao, Tong Yang, and Bin Cui. Cafe: Towards compact, adaptive, and fast embedding for large-scale recommendation models. *Proceedings of the ACM on Management of Data*, 2(1):1–28, 2024.