# HE-LRM: Encrypted Deep Learning Recommendation Models using Fully Homomorphic Encryption

Karthik Garimella*
*New York University*
*kg2383@nyu.edu*

Austin Ebel*
*New York University*
*abe5240@nyu.edu*

Gabrielle De Micheli
*LG Electronics USA, Inc.*
*gabrielle.demicheli@lge.com*

Brandon Reagen
*New York University*
*bjr5@nyu.edu*

***Abstract*—Fully Homomorphic Encryption (FHE) is an encryption scheme that not only encrypts data but also allows for computations to be applied directly on the encrypted data. While computationally expensive, FHE can enable privacy-preserving neural inference in the client-server setting: a client encrypts their input with FHE and sends it to an untrusted server. The server then runs neural inference on the encrypted data and returns the encrypted results. The client decrypts the output locally, keeping both the input and result private from the server. Private inference has focused on networks with dense inputs such as image classification, and less attention has been given to networks with sparse features. Unlike dense inputs, sparse features require efficient encrypted lookup operations into large embedding tables, which present computational and memory constraints for FHE.**

**In this paper, we explore the challenges and opportunities when applying FHE to Deep Learning Recommendation Models (DLRM) from both a compiler and systems perspective. DLRMs utilize conventional MLPs for dense features and embedding tables to map sparse, categorical features to dense vector representations. We develop novel methods for performing compressed embedding lookups in order to reduce FHE computational costs while keeping the underlying model performant. Our embedding lookup improves upon a state-of-the-art approach by $77\times$ [1]. Furthermore, we present an efficient multi-embedding packing strategy that enables us to perform a 44 million parameter embedding lookup under FHE. Finally, we integrate our solutions into the open-source Orion framework and present HE-LRM, an end-to-end encrypted DLRM architecture. We evaluate HE-LRM on UCI (health prediction) and Criteo (click prediction), demonstrating that with the right compression and packing strategies, encrypted inference for recommendation systems is practical.**

## 1. Introduction

Fully Homomorphic Encryption (FHE) is a cryptographic scheme that allows computations to be performed directly on encrypted data without requiring any decryption. By enabling secure computations in untrusted environments, FHE offers a powerful solution for privacy-preserving ap-

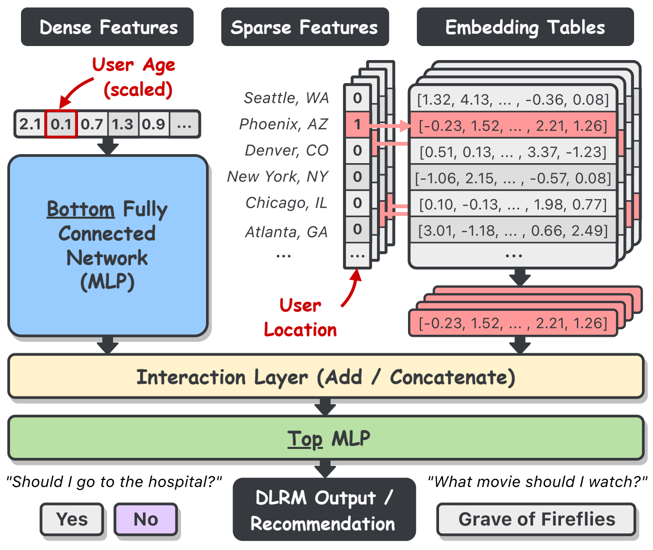

Figure 1. Overview of Deep Learning Recommendation Models (DLRM) [2]. DLRMs process both dense and sparse features: dense features pass through an MLP (Bottom) whereas each sparse feature retrieves an embedding vector from a lookup table. The dense output and embeddings interact before passing through a final MLP (Top) to make a prediction. **HE-LRM** utilizes Fully Homomorphic Encryption (FHE) to perform end-to-end encrypted inference of this model, maintaining data privacy throughout the entire computation.

plications in sensitive domains such as healthcare, finance, and personalized services. However, the high computational costs and the challenges of writing correct FHE programs limits its practicality to small-scale or narrowly scoped tasks.

Recent advances in FHE, from algorithmic improvements such as faster bootstrapping [3], [4], [5], [6], [7], [8], key-switching [9], [10], compiler optimizations [11], [12], [13], [14], [15], [16], [17], [18] and hardware acceleration [19], [20], [21], [22], [23], [24], have brought FHE closer to practical deployment for deep learning tasks. While prior works have explored FHE-based inference for simple neural networks or vision models, large-scale private recommendation models remain largely out of reach.

In particular, Deep Learning Recommendation Models (DLRMs), which enable personalized ranking and recommendation, pose unique challenges for privacy-preserving

*Authors contributed equally to this work.

deployment. These models have the particularity that they process two types of inputs: dense numerical features (e.g., user age, device characteristics) and sparse categorical features (e.g., user location, user IDs, item categories), which index into large embedding tables. We refer to Figure 1 for an illustration of a general DLRM model. Embedding tables are often tied to sensitive user attributes such as demographics, behavior, and preferences. This motivates the need for a privacy-preserving DLRM capable of performing inference over encrypted data.

This paper presents a fully homomorphic encryption solution for DLRMs using the RNS-CKKS FHE scheme [25] to enable inference on encrypted user data without compromising privacy. To the best of our knowledge, no solution exists that runs such models under FHE. As mentioned above, a key challenge in deploying (private) DLRMs is that embedding tables can grow prohibitively large, and performing efficient indexing under FHE is difficult. To address this, we propose an improvement over Kim et al.'s solution [1] for homomorphic lookup table evaluation. We perform a similar compression of embedding tables but overcome many of their limitations such as wasted slots in a ciphertext, high CKKS multiplicative depth, and even security considerations, while at the same time allowing for a similar exponential compression factor. Furthermore, we extend our technique to include a multi-embedding packing strategy to operate over multiple embedding tables in a parallel fashion.

We design our FHE-based recommendation model, which we refer to as HE-LRM inspired by the DLRM architecture. We implement HE-LRM in the open-source Orion framework [18], leveraging their state-of-the-art packing strategies, automatic bootstrap placement, and scale management. Moreover, the Orion framework allows us to develop our HE-LRM components directly in PyTorch, allowing for faster iteration and rapid prototyping throughout the development process.

We demonstrate the efficiency of our solution by running HE-LRM on two datasets. First, we test our model on the (smaller) UCI health dataset to predict heart diseases. In this case, we use the $x^2$ activation function rather than ReLU. We report an FHE latency of approximately 24 seconds with a validation accuracy of 85%. We also run HE-LRM on the Criteo Kaggle dataset for which the size of the embedding tables becomes critically large. We define a threshold to determine which embedding tables to compress (when the number of rows exceeds the threshold) and varying our threshold, we demonstrate FHE latency from 488 seconds down to 227 seconds when the compression ratio is $31180\times$, while the training loss and test AUC are minimally impacted by the compression. We also find the SiLU activation, which admits a lower-degree polynomial approximation than ReLU, significantly reduces the number of required bootstraps and improves efficiency. For example, the SiLU-based DLRM requires only 5 bootstraps and completes embedding lookups in 44 seconds, compared to 12 bootstraps and 51 seconds for its ReLU counterpart for a $10^2\times$ compressed model. Our main results are reported in Table 1.

TABLE 1. Summary of our experimental results when running HE-LRM on the UCI health dataset and the Criteo Kaggle dataset. Averaged over three runs.

| Dataset | # Parameters | FHE Latency (s) |
|---|---|---|
| UCI Health | 224 | 24.22 |
| Criteo Kaggle | 17 K | 227.7 |
|  | 144 K | 228.2 |
|  | 764 K | 230.0 |
|  | 9.1 M | 288.0 |
|  | 44.4 M | 488.9 |

Finally, we provide a discussion of the broader limitations and challenges of private recommendation systems, highlighting both the promise and open questions in deploying such models securely. Concretely, we make the following contributions.

1) HE-LRM, an end-to-end FHE-based DLRM model that performs encrypted inference on both dense and sparse features.
2) An FHE-friendly embedding compression technique to efficiently achieve an exponential compression factor for large embedding tables that outperforms state-of-the-art by $77\times$.
3) A multi-embedding packing strategy that leverages a block-diagonal structure optimized for baby-step-giant-step linear transformations.
4) An evaluation of our HE-LRM on the Criteo and UCI datasets, along with a detailed analysis of inference latency, accuracy, and memory footprint under FHE constraints.

## 2. Background

In this section, we provide a high-level overview of the RNS-CKKS fully homomorphic encryption scheme and also outline the DLRM architecture.

### 2.1. CKKS

CKKS [26], [27] is a SIMD style homomorphic encryption scheme that encrypts a vector of complex or real values into a ciphertext. This scheme relies on Ring Learning with Errors (RLWE) ciphertexts in $\mathcal{R}_Q^2$ for a given ring $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$, where $N = 2^k$, for $k > 0$, is the ring dimension and $\mathcal{R}_Q = \mathcal{R}/Q\mathcal{R}$ describes the ring $\mathcal{R}$ reduced modulo an integer $Q = \prod_{i=0}^{L} q_i$. The integer $L$ is known as the multiplicative depth and represents the maximum number of rescaling levels available before decryption fails. CKKS supports element-wise operations such as addition and multiplication as well as cyclic rotation. Based on the characteristics of the CKKS scheme, it becomes a natural choice for applications such as deep learning recommendation models, which accept real-valued vectors as input. We now describe some of the core operations used with CKKS.

**Encoding.** Consider a real (or complex) vector $\mathbf{x} \in \mathbb{C}^{N/2}$. This vector can be encoded into elements of $\mathcal{R}_Q$ using an

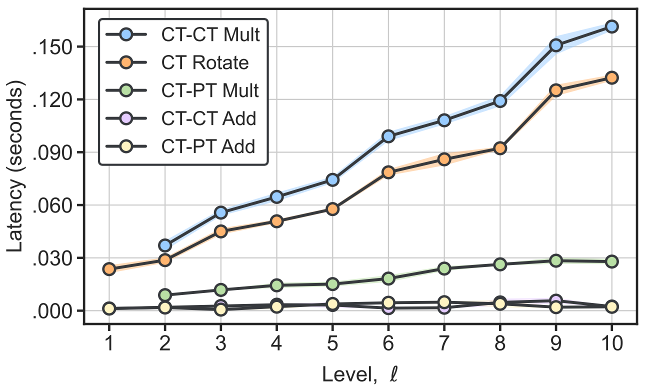

Figure 2. Latencies (single-threaded) of primitive homomorphic operations as a function of level averaged over 30 runs. Both ciphertext-ciphertext multiplication and ciphertext rotation require a compute and memory intensive key-switching operation. Multiplication time includes rescaling (for both cases) and key-switching (for CT-CT multiplication). Due to rescaling, CKKS multiplications require at least two limbs for both operands.

approximate inverse of a scaled complex canonical embedding. More precisely, one applies an inverse Fast Fourier Transform on the elements of $\mathbf{x}$ and scales each output by a scaling factor $\Delta$. Finally, each element is rounded to the nearest integer as encryption is performed over integers modulo $Q$. This step outputs a plaintext polynomial $\mathbf{m}(X)$ which packs $N/2$ complex values. These $N/2$ values are referred to as the available *slots* in a CKKS polynomial.

**Encryption** The plaintext polynomial $\mathbf{m}(X)$ can be encrypted into a ciphertext $(\mathbf{a}, \mathbf{b}) \in \mathcal{R}_Q^2$ with a given public key and the addition of some random noise.

**Addition.** CKKS supports element-wise plaintext-ciphertext and ciphertext-ciphertext additions. The resulting ciphertext after this operation corresponds to the SIMD addition of the underlying complex vectors.

**Multiplication.** Similar to addition, CKKS supports multiplication between either a plaintext and ciphertext or between two ciphertexts. Multiplication is followed by a rescaling procedure to avoid an exponential growth in the scaling factor. The resulting polynomial after this operation has coefficients in $\mathbb{Z}_{Q_{\ell-1}}$ instead of $\mathbb{Z}_{Q_\ell}$ where $Q_\ell = \prod_{i=0}^{\ell} q_i$ for $0 < \ell \leq L$ which corresponds to the consumption of a level in the chain of moduli.

**Rotation.** Cyclic rotations shift the elements of the input vector $\mathbf{x}$ by $0 < k < N/2$ slots. The resulting ciphertext after this operation corresponds to the same underlying vector with shifted elements by $k$ slots.

**Key-switching.** Key-switching is a standard operation in CKKS which converts a ciphertext encrypted under one secret key into a ciphertext that decrypts correctly under another secret key. This procedure is necessary after certain operations such as ciphertext-ciphertext multiplication or ciphertext rotation so that the result remains compatible with the decryption key. The transformation is done using a special switching key, which allows this change without decrypting the ciphertext.

**Bootstrapping.** In CKKS, a ciphertext can be *bootstrapped* to increase its number of available levels. This operation is necessary after the moduli chain has been depleted ($\ell = 0$) via multiplications and no more levels are available for further computations. Among all CKKS operations, bootstrapping is the most computationally expensive, taking roughly 20 seconds on a single-threaded CPU [28].

Figure 2 illustrates the latencies of the main CKKS homomorphic operations for both plaintexts (PT) and ciphertexts (CT) in terms of the level $\ell$.

## 2.2. Deep Learning Recommendation Model

Deep Learning Recommendation Models (DLRM) [2] are an example of a neural network-based recommendation model introduced by Facebook Research to handle both categorical (sparse) and continuous (dense) features. In order to process both types of inputs, the DLRM architecture combines multi-layer perceptrons (MLP) to process the continuous features and transforms categorical features into continuous embeddings through learned lookup tables. All processed inputs are then combined with some interaction operation. The model outputs a probability which corresponds to the likelihood of a user clicking an ad, a quantity referred to as click-through rate. More precisely, the DLRM architecture performs the following steps:

1) A *bottom MLP* processes the dense inputs. This MLP consists of a series of linear layers with ReLU activation functions.
2) The sparse (categorical) features are transformed into dense embedding vectors. One embedding table per categorical feature is used to retrieve these vectors.
3) The dense output from the bottom MLP and the embedding vectors are combined via an interaction operation, which can either be a dot product between all pairs of embedding vectors and the dense feature or a simple concatenation of the embedding vectors and the dense feature. In this work, we opt for the latter.
4) The concatenated features are then then fed into a *top MLP* that consists of a series of linear layers with ReLU activations.
5) The output of the top MLP is passed through a sigmoid function to produce a click-through probability.

We refer to [2] for more details about the algorithm and `https://github.com/facebookresearch/dlrm`. for the open-source implementation in both PyTorch and Caffe2. While originally developed for click-through rate prediction in targeted advertising, the DLRM architecture is flexible and can be adapted to process any input which contains both categorical and dense features and produce probability estimates for a wide range of applications. In this paper, we additionally apply the DLRM architecture to the the UCI Heart Disease dataset.

## 3. Threat Model

We assume a semi-honest threat model [29] in which an adversary faithfully takes part in the private inference protocol but may try to learn additional information from the messages they receive. This threat model (also known as honest-but-curious) is also assumed in a majority of prior works in private inference [18], [30], [31], [32], [33], [34].

We also assume that the client *knows the input dimension* to the neural network for which they query. In the context of CKKS and especially private CNN inference, this assumption has been implicit: the client encrypts their image into the slots of a CKKS ciphertext before sending them to server to be processed. The ordering of the pixels within the encrypted slots must be known by the server in order for the server to perform the encrypted computation correctly.

In this paper, we make this assumption explicit given that the inputs are no longer only dense features but also the sparse features used for the embedding tables. To this end, we assume that the client knows the input size to the Bottom MLP and the sizes (number of rows) of each embedding table. Crucially, the client does not know the embedding dimension (number of columns) or parameters of the embedding tables, the interaction method, or the parameters of the MLPs.

## 4. The Orion Framework

In this section, we explain our rationale for choosing the Orion framework and discuss our implementation of DLRM within it. Orion is a tool that translates deep neural networks written in PyTorch [35], a widely used deep learning library, into efficient FHE programs. Its high-level interface abstracts and automates the programming complexities that traditionally complicate FHE neural inference. This automation allows researchers to quickly iterate on ideas without dealing with low-level FHE implementation details. For most feed-forward neural networks, Orion automates SIMD data packing for linear layers, bootstrap placement, and scale management. Orion also handles polynomial approximations of activation functions and dynamically manages evaluation keys by loading and storing them as needed. We discuss data packing and bootstrap placement below.

**Efficient linear transforms:** In the clear, linear layers (e.g., fully connected, convolutional, FFN transformer blocks, etc.) dominate the inference latency of modern neural networks. The same trend holds when inputs are encrypted. Here, homomorphic linear transforms require many ciphertext rotations, and each rotation involves a costly key-switch operation as described in Section 2.1. Considerable effort has gone into reducing these costs, in part because they comprise a large portion of bootstrap latency in CKKS.

One notable method resulting from these efforts is the double-hoisting baby-step giant-step algorithm [28] (DH-BSGS). Double-hoisting amortized key-switching costs across many ciphertext rotations and BSGS significantly reduces the number of ciphertext rotations needed for matrix-vector products. Orion adopts this method for all linear layers in neural networks. For an in-depth analysis of encrypted matrix-vector products in RNS-CKKS, we refer the reader to Section 3 of Orion [18].

**Listing 1:** The Top MLP of DLRM within Orion.

```python
import orion.nn as on

class TopMLP(on.Module):
  def __init__(self, input_len):
    super().__init__()
    self.fc1 = on.Linear(input_len, 512)
    self.fc2 = on.Linear(512, 256)
    self.fc3 = on.Linear(256, 1) # decision

    self.relu1 = on.ReLU(degrees=[15,15,27])
    self.relu2 = on.ReLU(degrees=[15,15,27])
    self.sigmoid = on.Sigmoid(degree=31)

  def forward(self, x):
    x = self.relu1(self.fc1(x))
    x = self.relu2(self.fc2(x))
    x = self.sigmoid(self.fc3(x))
    return x
```

In this work, we utilize DH-BSGS linear transformations for all embedding table lookups. This optimization is central to the improvements over prior work and is further discussed in Section 5.

**Automatic bootstrap placement:** Orion formulates bootstrap placement as a shortest-path problem over a directed acyclic graph (DAG). Solving this problem determines the optimal ciphertext level for each Orion module that minimizes the overall encrypted inference latency. The runtime of linear layers is estimated using a cost model, and the aggregate latency accounts for the overhead introduced by necessary bootstrapping operations. Importantly, Orion supports bootstrap placement in modern, residual neural networks. These networks contain multiple paths from input ciphertext to output prediction. We find this property useful for DLRM, where inputs can either be sparse or dense, and thus take separate paths through the network. A benefit of extending PyTorch is that the infrastructure already exists to generate static computational graphs from neural networks. These graphs can be minimally post-processed to create the DAGs used in automatic bootstrap placement. Once the locations of bootstraps are known, their Orion modules are automatically inserted into the network graph and used in each encrypted inference. The concept of bootstrapping is entirely abstracted from the end user. We refer readers to Section 5 of Orion [18] for further details.

**Programmability:** We demonstrate Orion's programmability through our implementation of the top MLP within DLRM, shown in Listing 1. The implementation requires minimal changes from standard PyTorch: we simply replace each `nn` module with our equivalent `on` module and specify polynomial degrees to replace the `ReLU` activation function.

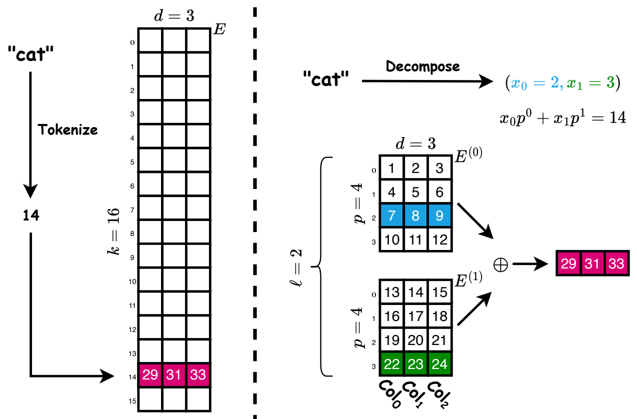

Figure 3. **Left**: A standard embedding lookup with a vocabulary size of $k = 16$ and embedding dimension $d = 3$. **Right**: Compressed tables can be learned [36] which decompose a large embedding into $\ell$ smaller tables of size $p \times d$ such that $p^\ell = k$. Input tokens are first mapped to coded tokens: $\mathbb{Z}_k \to \mathbb{Z}_p^\ell$. Each coded token retrieves a row from the compressed table and output rows are summed to produce a single embedding vector. [36] learns a mapping that is of size $O(k)$ whereas we employ a fixed bit-decomposition, detailed in Section 6.1.

## 5. Starting Point: CodedHELUT (ICML '24)

Given that Orion automatically supports the Bottom and Top MLPs in DLRM, we turn our attention to efficiently performing embedding lookups under FHE. Our starting point is the Coded Homomorphic Evaluation of Lookup Tables (CodedHELUT) algorithm from [1], which we describe in this section and improve upon in Section 6. We note that efficient FHE embedding table lookup extends beyond DLRMs and may also be of interest for encrypted language model inference since LLMs also require embedding lookups.

### 5.1. Embedding Tables

As discussed in Section 2.2, the categorical (sparse) features are represented by embedding vectors stored in embedding tables. While a cleartext DLRM can perform standard embedding lookups directly, the homomorphic setting requires careful design of an efficient retrieval algorithm.

Embedding tables in DLRMs are parameterized as matrices $E \in \mathbb{R}^{k\times d}$ where $k$ is the number of unique categories and $d$ is the hidden or embedding dimension. An example embedding table can be seen in the left of Figure 3. For standard cleartext computations, the inputs to the embedding table are simply indices $i \in \mathbb{Z}_k$, which are used to retrieve rows (i.e., $E[i]$), from the embedding table for further processing by the DLRM. However, this process of extracting rows using indices is challenging in the context of CKKS where the only available operations are SIMD addition, multiplication, and cyclic rotation of encrypted vectors.

A straightforward FHE-friendly solution to this lookup problem is to treat the embedding lookup as a linear transformation by an encrypted one-hot encoded index. First, define

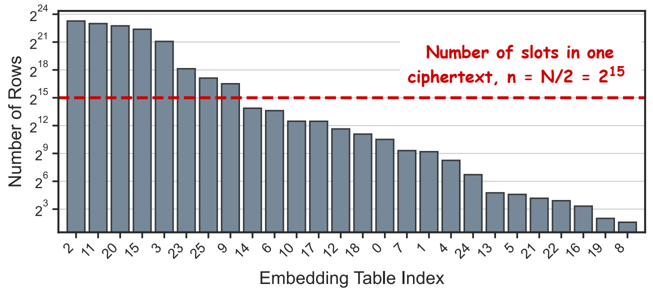

Figure 4. Embedding tables for the 26 categorial features in the Criteo dataset sorted by size (number of rows). There are a total of 33.8 million rows for all features.

the one-hot encoding function as $\mathsf{OHE} : \mathbb{Z}_k \to \{0,1\}^k$ such that for any $i \in \mathbb{Z}_k$, we have

$$\mathsf{OHE}(i) = (e_0, \ldots, e_{k-1}) \text{ where } e_j = \begin{cases} 1 & \text{if } j = i \\ 0 & \text{otherwise} \end{cases}$$

Then, by encrypting this basis vector $\mathsf{OHE}(i)$ under FHE, the server can perform the vector-matrix multiplication $\mathsf{OHE}(i) \cdot E$ homomorphically. This encrypted computation would be equivalent to the cleartext index lookup of $E[i]$. The primary drawback of this method is that each one-hot encoding requires $k$ slots of a CKKS ciphertext where only a single element within these $k$ slots is non-zero. Figure 4 shows that realistic DLRMs exhibit large embedding table sizes. For a DLRM tailored to the Criteo dataset, the naïve, one-hot encoding for each embedding table would require 33.8 million slots. And for standard FHE parameters with 128-bit security with a slot count of $2^{15}$, the sparse input for Criteo would require approximately 1000 CKKS ciphertexts.

### 5.2. Compressed Tables with CodedHELUT

To avoid requiring $k$ slots for an embedding table containing $k$ rows, prior work [1] uses compressed embedding tables learning through Deep Compositional Code Learning [36]. In this setup, a $k$-sized embedding table is decomposed into $\ell$-many, $p$-sized tables such that $k = p^\ell$. In this way, an index $i \in \mathbb{Z}_k$ is first mapped to a sequence of $\ell$ tokens $(i^0, \ldots, i^{\ell-1}) \in \mathbb{Z}_p^\ell$ and each coded token retrieves a row from each of the smaller tables. Finally, the output embeddings from each of the $\ell$ tables is summed to produce the final output embedding vector. This compression technique reduces a table by an exponential factor (from $k = p^\ell$ rows to only $p\ell$ rows) and experiments over the GlovE and GPT-2/Bert embedding tables display its efficacy in reducing table sizes (see Section 5 in [1]). Figure 3 shows this technique: an uncompressed table of size $k = 16$ is reduced to $\ell = 2$ embedding tables of size $p = 4$, effectively halving the total embedding table size.

The prior work of [1] leverages this compression technique, but performs the one-hot encoding server-side. They do this by building an encrypted indicator function (EIF), which effectively transforms a sequence of coded tokens into their one-hot equivalent. This process is decomposed

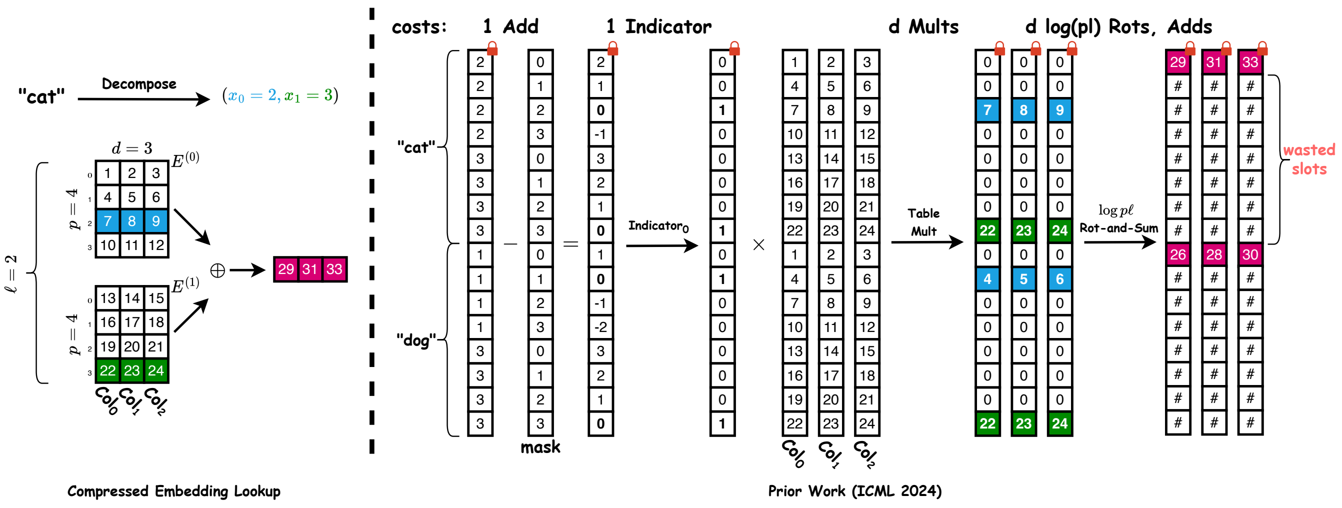

Figure 5. **Left**: An example of compressed embedding tables with a total of $p\ell = 8$ rows whereas the original embedding table (not shown) had $p^\ell = 4^2 = 16$ rows. A token (e.g., "cat") is decomposed into a sequence of coded tokens which are used for indexing into each compressed table, followed by a summation to produce a single embedding vector of size $d = 3$. **Right**: Prior work [1] performs this compressed lookup homomorphically by utilizing $p\ell$ slots per ciphertext and performing one homomorphic $\mathsf{Indicator}_0$ function followed by $d$ multiplications with the concatenated columns of the compressed tables. These operations are then followed by $d \log(p\ell)$ rotations and summations (see Table 4 of [1]). Rotation and summation *within* a ciphertext produces wasted slots filled with invalid data. In this example, two separate tokens ("cat" and "dog") fit into a single 16-slotted ciphertext.

into two steps. First, a squaring method approximates a standard indicator function $\delta_a : \mathbb{Z}_p \to \{0, 1\}$ for $a \in \mathbb{Z}_p$ by computing,

$$\texttt{SqMethod}^a_{r,p}(x) = \left(1 - 2\frac{(x-a)^2}{p^2}\right)^{2^r}$$

for a well-defined $r$ chosen large enough to ensure a reasonable approximation bound (see [1, Theorem 3.1]). Second, a cleaning function `Cleanse` given as a degree-3 polynomial is applied to the output of `SqMethod` to efficiently round the value to 0 or 1. Thus, the EIF is a composition of the two previous steps,

$$\texttt{Indicator}_a(ct) = \texttt{Cleanse}^s \circ \texttt{SqMethod}^a_{r,p}(ct),$$

where $ct$ is a given ciphertext and $s$ is the number of calls to `Cleanse` for which an upper bound is given in [1, Proposition 3.3]. Given the aforementioned EIF, [1] provides two algorithms (with and without compressed embeddings) for lookup table evaluations with encrypted data. The full algorithm from [1] is shown in Figure 5. We detail each of the steps. Step 1: transform the initial token into the coded tokens. In this case, the token "cat" (14) is transformed into coded tokens $(2, 3)$. Step 2: replicate each of the $\ell$ coded tokens $p$ times into the first $p\ell$ slots of a CKKS ciphertext and send to the server. In Figure 5, we have $p = 4$ and $\ell = 2$ and therefore place our replicated coded tokens in the first 8 slots of the ciphertext. Step 3: the server subtracts the ciphertext with a plaintext mask and performs $\texttt{Indicator}_0$, effectively performing the one-hot encoding server-side. Step 4: multiply the encrypted one-hot encoding with each concatenated columns of the embedding tables (called TableMult). Step 5: perform a logarithmic number of rotations and summations to compute the final embedding vector. We refer to [1, Algorithm 3 and Algorithm 4] for more details. We implement Kim et al.'s [1] solution's directly in Orion (see Listing 2) and run all experiments on the same machine in order to make a fair comparison.

## 5.3. Limitations of CodedHELUT

We address several of the limitations of the CodedHELUT method presented in [1].

**Slot Utilization:** The CodedHELUT method requires $O(\log p\ell)$ rotations for each output ciphertext to compute the desired embedding vector as shown in Figure 5. Their approach uses the standard logarithm-based rotation-and-summation, which we call the Rot-Sum algorithm (see Figure 4 of HECO [16] for a diagram of Rot-Sum). There are two issues with using the Rot-Sum method. First, the slot occupancy for $p\ell$ must necessarily be inflated to the closest power of 2. As an example, summing 6 slots with Rot-Sum requires padding up to 8 slots with two zeros and applying Rot-Sum to all 8 slots. Second, a consequence of applying Rot-Sum *within* a ciphertext is that most slots are filled with undesired partial sums or invalid data. We visualize this in Figure 5 where, for every $p\ell = 8$ slots, there are $p\ell - 1 = 7$ slots that contain invalid data. In contrast, our BSGS-based embedding lookup supports arbitrary dimensions and does not waste any slots by summing the coded tables *across* ciphertexts rather than *within* ciphertexts.

**Output ordering**: While the CodedHELUT algorithm produces the correct embedding, this particular embedding vector is *sharded* across $d$ ciphertexts where $d$ is the embedding vector dimension. This can be seen in Figure 5 where the desired embedding vector (e.g., the vector $[29, 31, 33]$) is fragmented across the three output ciphertexts. In order to

**Listing 2:** Indexing embedding tables using EIF.

```python
import torch.nn as nn
import orion.nn as on

class Cleanse(on.Module):
  def __init__(self, iters):
    super().__init__()
    polys = []
    for i in range(iters): # -2x^3 + 3x^2
      poly = on.Activation(coeffs=[-2, 3, 0, 0])
      polys.append(poly)
    self.polys = nn.Sequential(*polys)

  def forward(self, x):
    for poly in self.polys:
      x = poly(x)
    return x

class EIF(on.Module):
  def __init__(self, p, r, iters):
    super().__init__()
    self.cleanse = Cleanse(iters)
    self.sq = on.SqMethod(p, r)

  def forward(self, x):
    x = self.sq(x)
    x = self.cleanse(x)
    return x

class Embedding(on.Module):
  def __init__(self, p, l, r, iters, dim):
    super().__init__()
    self.EIF = EIF(p, r, iters)
    self.query = on.Table(l, p, dim)

  def forward(self, x):
    x = self.EIF(x)
    x = self.query(x)
    return x
```

perform the subsequent concatenation or linear transformation in either DLRMs or LLMs, this fragmented embedding vector must be consolidated into contiguous slots. Performing this consolidation requires 1) further rotations to align the embedding elements and 2) masking to remove the junk data from the wasted slots, which increases level consumption. On the other hand, our BSGS-based method produces the correct embedding vectors in contiguous slots while consuming only a single level.

**Level Consumption**: Beyond the levels consumed to consolidate the output embedding vector, the indicator function incurs a multiplicative depth of $2 + r + 2s$ (see Section 3.3 of [1]). Based on both the FHE parameters and the iteration parameters $r$ and $s$, the indicator function may require an expensive bootstrap operation. Indeed for a 128-bit secure FHE configuration, we find that we must judiciously set both the ciphertext and auxiliary moduli to ensure that the Indicator function does not *overflow* the available levels. Conversely, our embedded lookup only consumes one level regardless of the embedding size.

**Security Issue**: The embedding compression technique used by CodedHELUT requires that the server learns both the

**Listing 3:** Our compressed embedding in Orion.

```python
import orion.nn as on

class CompressedEmbedding(on.Linear):
  def __init__(self, p, l, dim):
    super().__init__(p*l, dim, bias=False)

  def populate_table(self, data):
    self.weight.data = data

  def forward(self, x):
    return super().forward(x)
```

compressed embedding tables as well as the mapping between original tokens to coded tokens (see Figure 3 of [1]). Moreover, it is the client's task to first map the original token to the coded tokens and then encrypt their coded tokens. There are two possible protocols that enable their compressed table technique which we now detail.

*Server-side API*: In this scenario, the client must query the server to transform their original token into the coded tokens. This compromises the client's privacy as the client must send their original token in the clear. Therefore, a server-side API leaks a portion of the client's input directly to the server.

*Client-side API*: the mapping is sent to the client who then generates their coded tokens locally. However, the particular mapping utilized by [1] ensures that semantically similar input tokens map to a similar set of coded tokens. Thus the client will learn some semantics about the rows of the embedding tables and their relationships. Second, each mapping must itself be a table with the same number of rows as the original embedding tables. In the client-side API, the client is now responsible for storing these mapping tables. In contrast, our mapping (Section 6.1) is deterministic, requires no additional storage, and can be done locally without knowing anything other than the embedding table sizes.

## 6. HE-LRM Architecture

We now describe our end-to-end FHE-friendly DLRM architecture, which we refer to as HE-LRM. Each HE-LRM inference takes as input an encryption of the concatenated dense and sparse features and returns an encrypted, unnormalized logit to the client. We discuss our method for compressing embedding tables, our protocol for parallelizing multiple embedding tables, and implementation details within the Orion framework.

### 6.1. Simplified Compressed Embedding Tables

Recall that prior work [1] built their CodedHELUT algorithm to integrate with the compositional embedding tables from [36], a method that compresses an embedding table of size $k \times d$ into $\ell$ compressed embedding tables, each of size $p \times d$ such that $k = p^{\ell}$. This technique learns both the compressed embedding tables and the mapping from

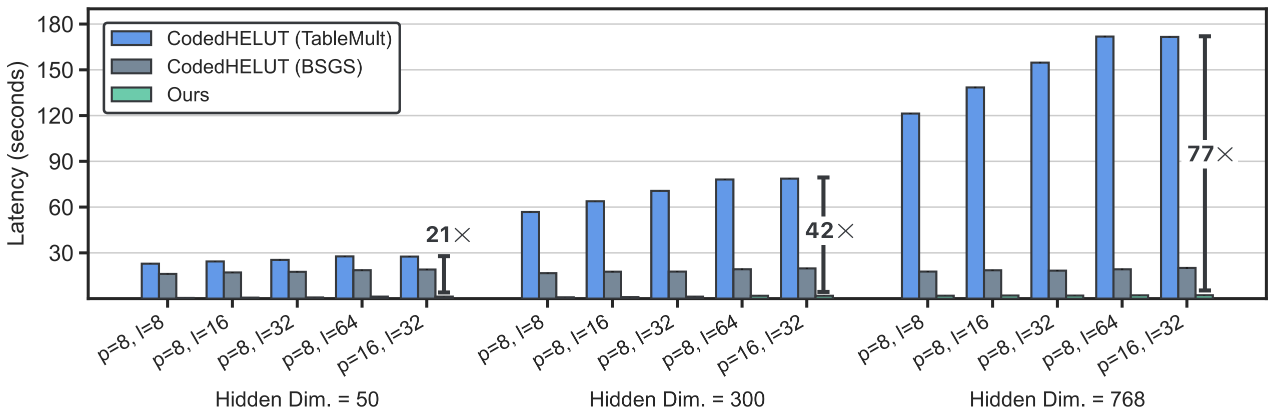

Figure 6. A comparison of encrypted embedding lookups using a fixed set of FHE parameters (see `configs/resnet.yml` in Orion) for different settings of $k = p^{\ell}$ and embedding dimensions, $d$. The CodedHELUT method proposed in [1] first uses the Encrypted Indicator Function (EIF) to perform the one-hot encoding server side. This function requires a bootstrap operation given the multiplicative depth of EIF. For the embedding lookup stage, swapping out their proposed TableMult algorithm with a double-hoisted BSGS linear transformation already reduces runtime as TableMult requires separate rotations for every hidden dimension, $d$. Our proposed solution performs the one-hot encoding client-side while still requiring the same slot count as the prior method and directly leverages DH-BSGS. Moreover, our embedding lookup has a multiplicative depth of one, regardless of the embedding size.

input tokens $i \in \mathbb{Z}_k$ to a sequence of tokens in $\mathbb{Z}_p^{\ell}$, and this mapping is stored as its own lookup table.

Given the security considerations with regards to querying or storing the mapping tables (see Section 5.3), we instead opt for a much simpler compression technique that can been seen as a generalization of the Quotient-Remainder (QR) method from [37]. The QR method decomposes a single $k$-sized embedding table into two tables: a Quotient and Remainder table by computing $(i \bmod q, i\%q)$ for some $q < k$. These two QR tables are then trained end-to-end.

We extend the QR method by instead performing a *bit decomposition* of an original input token $i \in \mathbb{Z}_k$ into base $p$ to form a tuple of tokens in $\mathbb{Z}_p^{\ell}$. This process still produces a maximum of $\ell = \lceil log(k,p) \rceil$ tokens and allows a similar exponential compression factor of an embedding table of size $k \times d$ into $\ell$-many $p \times d$ embedding tables.

We use Figure 5 (left) as a concrete example of our compression technique. Suppose that we have an embedding table with $k = 16$ rows and $d = 3$ as the hidden dimension. By choosing a base $p = 4$, each input index $i \in \mathbb{Z}_{16}$ can be represented as a tuple in $\mathbb{Z}_4^2$ given that $\ell = 2 = \lceil \log(16, 4) \rceil$. In Figure 5, the original input token $i = 14$ becomes a tuple of tokens $(i_0, i_1) = (2, 3)$. Each coded token then retrieves a row from its corresponding embedding table, and all retrieved output vectors are summed to produce a single embedding vector of size $d$.

It is important to note that unlike [36], our compression technique does not require learning or storing the mapping from an index to a set of coded tokens. Rather, our mapping is a deterministic function that can be done entirely by the client given that they know the size of the compressed embedding tables, which is encompassed by our threat model. Performing this one-hot encoding prior to encryption allows us to forgo the encrypted indicator function and directly treat the embedding layer as a linear transformation. Listing 3 shows our embedding lookup which inherits directly from Orion's `Linear` module and leverages the double-hoisted baby-step giant-step algorithm. Figure 6 compares the CodedHELUT+TableMult, CodedHELUT+BSGS, and our solution in terms of latency for various hidden dimensions. Replacing TableMult with a linear transformation reduces the runtime since less homomorphic rotations are required. Furthermore, our method outperforms CodedHELUT for all considered hidden dimensions with up to a 77× speedup for the largest dimension of 768.

We train DLRM models on the Criteo Kaggle 7-Day Dataset (https://www.kaggle.com/c/criteo-display-ad-challenge/data) which, as seen by Figure 4, has a maximum embedding table size of approximately 10 million rows. For each training run, we set a maximum embedding table size; any embedding table larger than this threshold are compressed using our generalized QR technique by selecting a bit decomposition base $p$. We present our loss curves, test AUC measurements, and FHE latencies in Section 7.

To perform the compressed embedding lookup homomorphically, we perform the one-hot encoding of each coded token and concatenate these one-hot vectors into a single ciphertext. Then, by concatenating and transposing the compressed tables, we may apply the standard BSGS linear transformation to produce the correct embedding in contiguous slots. This process is shown in Figure 7 for embedding $E_0$ that has been compressed into two tables.

## 6.2. Parallel Embedding Table Lookup

Our discussion in the previous sections was limited to performing an encrypted lookup into a single (compressed or uncompressed) embedding table. However, state-of-the-art DLRMs usually contain more than one sparse feature given the problem specification. Indeed, the UCI Healthcare dataset [38] and the Criteo Kaggle dataset which we target

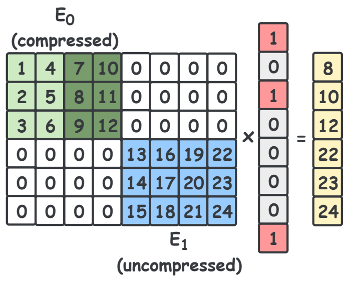

Figure 7. Our embedding lookup technique. The first embedding table $E_0$ has been compressed into two tables (light and dark green). The one-hot coded tokens for $E_0$ are packed into contiguous slots and the corresponding matrix-vector product extracts the appropriate embedding. Furthermore, we pack multiple embedding tables diagonally into a single weight matrix. In this example, two embedding lookups are performed simultaneously using a single homomorphic matrix-vector product. This diagonal packing aligns well with the diagonal-based BSGS matrix-vector products used in Orion.

require 8 and 26 embedding tables, respectively. Furthermore, both datasets contains embedding tables of different sizes (different number of rows per categorical feature). Under the constraints of FHE, we are posed with the following question: how can we efficiently perform encrypted lookups when we have multiple embedding tables of different sizes?

A straightforward solution to this multi-embedding table case is to simply encrypt each compressed index *separately* and perform the methods discussed before. However, this method would require one CKKS ciphertext per embedding table and may be wasteful when compressed tables have a total size less than the number of available CKKS slots. For our use cases with both UCI and Criteo, we find that an aggressive compression ratio induces this exact scenario.

To this end, we propose a multi-embedding packing strategy which places unique embedding tables *diagonally* across a standard weight matrix by starting each subsequent embedding table at the bottom right corner of the previous embedding table. This packing strategy is akin to calling: `torch.block_diag(*list_of_embeddings)` (we use this exact command in our implementation). An example of our packing strategy can be seen in Figure 7 where exactly two unique feature embedding tables are stored in a single weight matrix. The client simply one-hot encodes each index for each table (again, either compressed or uncompressed) and concatenates these one-hot vectors into a single vector, which can then be encrypted. The resulting matrix-vector product precisely extracts the correct columns of the packed embedding matrix and places the embedding vectors into contiguous slots within an output ciphertext.

Importantly, this diagonal multi-embedding packing strategy aligns well with diagonal-based BSGS matrix-vector product strategies discussed in Section 4. Moreover, diagonally aligning embedding tables incurs little FHE overhead as long as the sum of the embedding dimensions does not exceed the slot count.

## 6.3. Implementation Details

In this section, we discuss some of the sharper bits of ensuring proper execution of the HE-LRM architecture within Orion. We refer the reader to Figure 8 for our full implementation of HE-LRM.

**6.3.1. HE-LRM Inputs.** Orion generalizes ciphertexts to ciphertensors objects that can encrypt more elements than the slot count set by the FHE parameters. Their method works by partitioning the input data into slots-many chunks where each chunk is encrypted into a separate ciphertext. This naturally enables large-scale matrix-vector products that require more than one input or output ciphertext.

For HE-LRM, we one-hot encode each embedding token. For uncompressed tables, this is the standard one-hot encoding. And for compressed tables, the token is first bit decomposed as discussed in Section 6 and each bit is one-hot encoded and then concatenated. Finally, the dense input and *all* one-hot encodings are concatenated together to form a single ciphertensor object that is sent to the server.

**6.3.2. Server-Side Extraction.** The server must then extract both the dense and sparse features from the received ciphertensor object. We implement a custom extraction module in Orion that uses 1) plaintext masks to isolate the dense and sparse features and 2) performs homomorphic rotations to align the data within the ciphertensors accordingly. This extraction process only consumes one multiplicative level.

**6.3.3. Concatentation Layer.** After both the Bottom MLP and embedding table lookups, the outputs must then be concatenated into a single ciphertensor object to be processed by the Top MLP. We perform this concatenation via a linear transformation applied to both the Bottom MLP and extracted embeddings. The linear transformation aligns the data separately before an `on.Add` module combines the two dense and embedding vectors together in the correct slots. Both linear transformations only have one non-zero diagonal: an offset identity matrix to align data appropriately while preserving shape semantics.

## 7. Results

In this section, we report our results and discuss our findings for both training FHE-amenable DLRMs and performing end-to-end FHE inference of HE-LRMs.

## 7.1. Experimental Setup

**Hardware**: We conduct all of our experiments on an Intel Xeon Gold 5218 processor running at 2.30GHz with 64 CPU cores and 512 GB of RAM. We find this amount of RAM necessary to run our largest compressed DLRM models under FHE. We train all DLRMs on NVIDIA 3090 GPUs, and experimental results (training losses, AUCs, and FHE latencies) are all averaged over 3 runs.

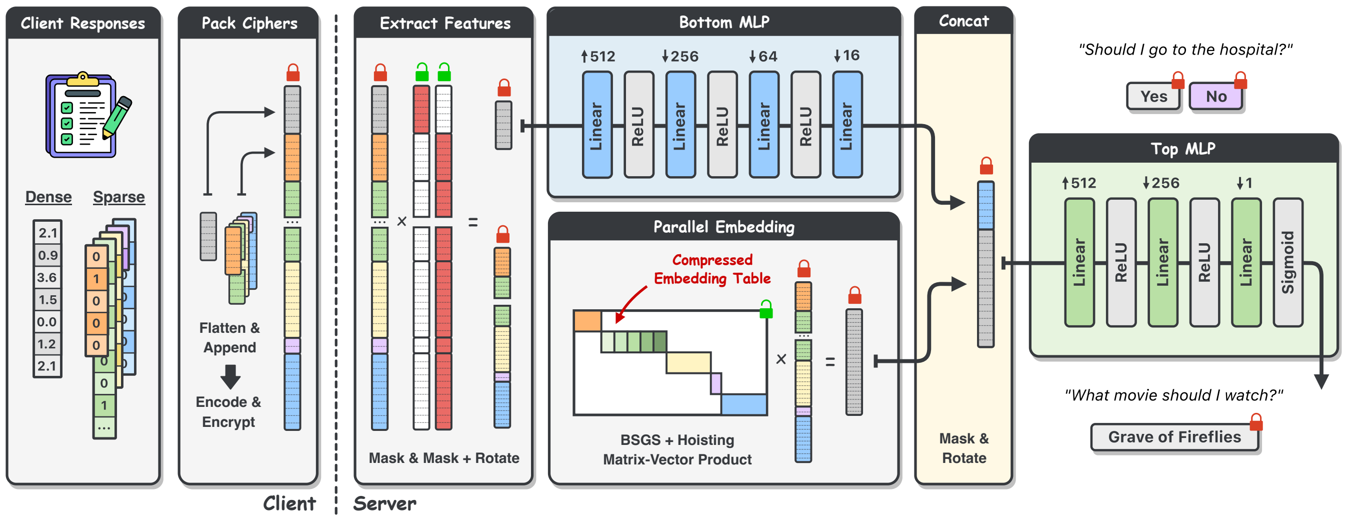

Figure 8. The HE-LRM architecture. Beginning from the left, the client first records both their dense and sparse features for some outsourced recommendation system (e.g., a medical inquiry). The client then one-hot encodes (either compressed or uncompressed) their sparse features, appends these one-hot vectors to the dense features, and encrypts the resulting vector. The server performs the appropriate homomorphic operations to extract the dense and sparse features into two separate ciphertexts. The dense features are processed through the Bottom MLP whereas the one-hot sparse features are used as input to the diagonally-packed embedding tables in order to perform parallel embedding table lookups. The resulting vectors are concatenated and are further processed through a Top MLP before sending the encrypted logit back to the client. We implement this entire process in the Orion framework.

**Software**: Our implementation builds upon the CAFE framework (https://github.com/HugoZHL/CAFE), which provides an optimized DLRM implementation. We implement the generalized QR decomposition technique within CAFE. We use the hyperparameters set by their repository. We port all DLRM models directly into the Orion framework and implement all necessary components (extraction, concatenation, embeddings). Unless stated otherwise, we choose an FHE parameter set that enables bootstrapping while maintaining 128-bit security. The exact parameters can be found in `configs/resnet.yml` in Orion.

**Datasets**: We evaluate HE-LRMs on two datasets: the UCI Heart Disease (id=45) and Criteo 7-day click-through datasets. The UCI Heart Disease dataset is used to predict the detection of heart disease based on 13 attributes (5 dense and 8 sparse). Features include age, resting blood pressure, and maximum achievable heart rate. There are 303 training samples; we use 80% of the samples for training set and 20% for the validation set. This dataset serves as a basis for testing the validity of our methodology as well as the latency of small DLRMs, in a privacy-sensitive setting. The Criteo dataset includes 13 continuous features (normalized integers) and 26 categorical features where each feature has been anonymized. There are approximately 45 million training samples and this dataset is used to predict the click-through-rate for a provided advertisement. The first 6 days of data are used for training while the seventh day serves as the validation set. We are able to train a DLRM on the Criteo dataset from scratch in about 2 hours on a single 3090 GPU.

TABLE 2. COMPRESSION RATIO AND RESOURCE REQUIREMENTS FOR DIFFERENT EMBEDDING TABLE THRESHOLDS.

| Threshold | Compression Ratio | Slots Needed | Peak RAM (GB) |
|---|---|---|---|
| 500 | 31180× | 1096 | 64 |
| 5000 | 3746× | 9027 | 70 |
| 50000 | 707.1× | 47759 | 74 |
| 500000 | 59.28× | 569545 | 140 |
| 5000000 | 12.18× | 2772109 | 320 |

## 7.2. Training Compressed DLRMs

To enable efficient FHE operations on large embedding tables, we implement our generalized bit-decomposed embedding table compression strategy within the CAFE framework and train DLRMs on the Criteo dataset. First, we set a particular embedding table size *threshold*. For each sparse feature in the Criteo dataset, we compare the corresponding embedding size (i.e., number of rows) with our threshold. If the number of rows is larger than the threshold, we apply our bit-decomposed compression strategy to this particular embedding table. Otherwise, we employ the full embedding table. In all of our experiments, we set the base of the decomposition to be $p = 4$.

Similar to prior work [39], we define the Compression Ratio (CR in Table 2) as the ratio of the total number of embedding rows in the full DLRM to the total number embedding rows in a compressed model. We vary the threshold to roughly track an order of magnitude of increases in the compression ratio. Table 2 shows our chosen thresholds and their exact CR. An aggressive threshold of 500 rows

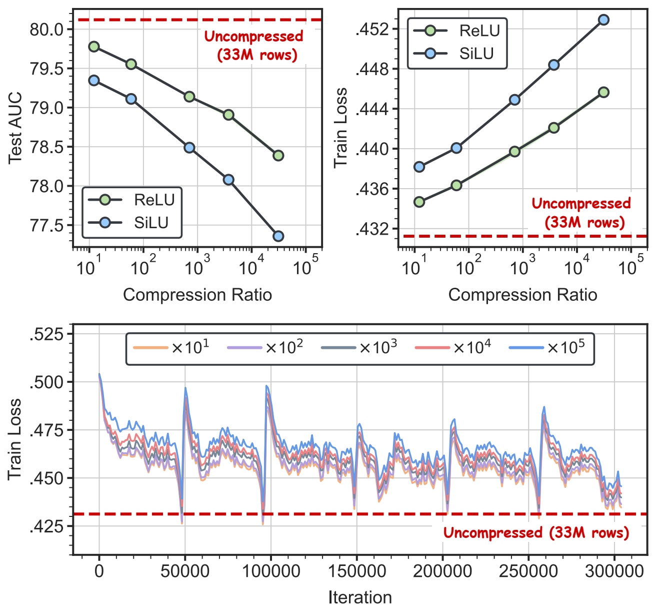

Figure 9. Training curves and performance trade-offs for compressed DLRM models across different compression ratios. **Top Left**: Final test AUC shows degradation with increased compression for both ReLU and SiLU activations. **Top Right**: Final training loss comparison between activation functions. SiLU routinely performs worse than ReLU-based DLRMs. **Bottom**: Training loss curves for our compressed DLRMs with ReLU. Models that have higher compression rates exhibit slightly elevated loss throughout the training process. Given the time-based nature of the Criteo dataset, all loss curves follow an expected diurnal pattern.

compresses a majority of the 26 sparse features for Criteo and reduces the entire embedding size by $31180\times$. Table 2 also shows the peak RAM usage when running FHE inference for each configuration as measured by `htop` [40].

Figure 9 (bottom) displays training runs for each compression threshold configuration, We find that all models follow similar training dynamics and fall naturally on the loss spectrum based on the compression ratio: a less (more) compressed model has a slightly lower (higher) loss over the entire training run. The diurnal trend in the loss is expected. The Criteo dataset is processed in-order over the course of 6 days for training, and each new day initially increases the loss before the loss begins to decrease again.

We also train each DLRM configuration with the SiLU activation function rather than the standard ReLU activation function. We do this since SiLU is a smoother activation function and therefore is easier to approximate in FHE using a composition of polynomials when compared to ReLU. However, we find that for each configuration, the SiLU networks exhibit both lower test AUC and higher training losses. As we will show in the following section, this introduces a latency-performance tradeoff in which SiLU models have much lower FHE latencies but perform worse out of distribution.

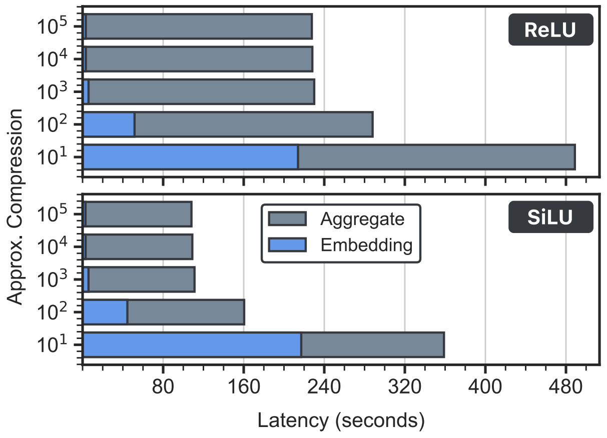

Figure 10. End-to-end (aggregate) FHE latency for a single Criteo sample passed through our compressed HE-LRMs averaged over three runs. Exact compression ratios and peak RAM usage are listed in Table 2.

## 7.3. HE-LRM Latencies

**UCI Heart Disease**: We first apply our HE-LRM architecture to the underlying DLRM model trained for the UCI Heart Disease dataset. This dataset has 5 dense features and 8 sparse features with sizes $[2, 4, 2, 3, 2, 3, 4, 3]$ giving an input ciphertext where 28 slots are needed. Given the size of the input, we scale down the DLRM model and use the $x^2$ activation function rather than ReLU. Nonetheless, this model achieves a validation accuracy of 85% and serves as a useful testing framework for our implementation.

We are able to fit all embedding tables into a single matrix-vector product by using our multi-embedding diagonal packing technique described in Section 6.2. Averaging over three runs, we achieve an end-to-end latency of 24.22 seconds for this model. Given our FHE parameter set, this model requires exactly one bootstrap which takes up 75% of the runtime.

**Criteo Kaggle**: Figure 10 presents our key end-to-end FHE latency results for a single Criteo input across our compressed DLRM configurations averaged over three runs. We highlight two key observations from our results.

*Observation 1*: All highly compressed models exhibit similar FHE latencies. The three most aggressively compressed models ($\approx 10^3\times$ to $10^5\times$ compression ratio) all exhibit similar latencies of approximately 230 seconds for the ReLU based models. This similarity is due to two factors. First, our multi-embedding parallel packing scheme allows us to perform a single-ciphertext linear transformation for two smallest models. Both $10^4\times$ and $10^5\times$ have 512 non-zero diagonals in the plaintext blocked embedding matrix whereas the $10^3\times$ model has 1024 non-zero diagonals and requires two ciphertexts to hold the one-hot inputs. Thus, the embedding lookup takes 3-5 seconds for these three highly compressed models. Second, apart from the embedding size, the DLRM architecture is the same for all models, meaning that all FHE operations are identical outside of the embedding lookup. Indeed, Orion places the bootstrap operations in the same locations for all three networks.

On the other hand, the larger models ($10^2\times$ and $10^1\times$) require many ciphertexts to store all one-hot coded tokens. Table 2 shows the total number of slots required by these models: the $10^2\times$ and $10^1\times$ compressed models require 18 and 85 ciphertexts to hold all one-hot encoded inputs, respectively. This inflates the matrix-vector product latency; for example the largest model takes 213.9 seconds to perform the embedding lookup.

*Observation 2*: Changing the activation function impacts the FHE level management policy. In more detail, the SiLU activation is a smoother function when compared to ReLU and can be approximated by a lower degree polynomial. This lower degree approximation, in turn, requires overall less bootstraps for HE-LRM and affects the overall level management policy output by Orion. Concretely, the SiLU based model requires just 5 bootstrap operations whereas the ReLU based models require 12 bootstraps. Furthermore, the level management policy constructed by Orion performs the embedding layers in the SiLU networks at $\ell = 3$ whereas the embedding is performed at $\ell = 4$ for the ReLU models. For this reason, the $10^2\times$ model takes only 44 seconds for the SiLU-based DLRM, but takes 51 seconds for the ReLU-based DLRM.

## 8. Limitations and Considerations

In this section, we highlight several practical limitations that emerged during our exploration of secure recommendation systems. Unlike image recognition models, which have been the primary focus of previous private inference research, recommendation systems present more subtle and nuanced challenges. In some cases, these issues raise fundamental questions about the viability of deploying such systems in practice. To better describe these limitations, we divide recommendation systems into two categories: static systems and dynamic systems.

Static recommendation systems do not require a *history* of prior user interactions to accurately predict the current outcome. For this reason, we consider static systems a natural extension of traditional image recognition models, differing mainly in their ability to handle sparse, categorical inputs. For example, the UCI Heart Dataset DLRM is a static recommendation system, since it requires only the user data provided within a single inference to make accurate predictions. These characteristics make static systems well-suited for private inference.

In contrast, dynamic recommendation systems progressively learn the dense and sparse features needed for accurate predictions through repeated user interactions over time. Typical examples include personalized advertisements on social media or product recommendations on e-commerce sites. These systems naturally arise when companies have both financial incentives and legal permissions to keep extensive histories of user interactions. As a result, most recommendation systems deployed today fall into the dynamic category. This unfettered access to cleartext user interaction histories significantly complicates the adoption of private inference. After all, why would companies willingly put a primary source of their revenue at risk without external pressure (e.g., GDPR or CCPA)? Below, we discuss several potential solutions, highlight their inherent trade-offs, and raise additional questions to further explore this issue.

*“Why not anonymize user data instead?”* True anonymization makes it impossible for the server to provide targeted recommendations, despite being able to train a capable model. Pseudo-anonymization exists in practice, however still requires a link between the user and their data.

*“What if I encrypt just my important data?”* It is likely that the server can infer the contents of the encrypted data based on user history (e.g., search history can often reveal a user’s location without location services enabled).

*“What if the embedding table(s) are public?”* In the context of DLRMs, public embedding tables leak information about all users of the recommendation service. In the context of LLMs, embedding tables are considered as part of the network parameters and are generally kept private.

*“How can servers practically deliver personalized recommendations if the inference results remain encrypted?”* Servers can deliver personalized recommendations from encrypted results using auxiliary protocols such as private information retrieval (PIR).

*“What are the drawbacks server-side for enabling encrypted recommendations?”* Privacy-preserving techniques such as FHE and PIR will allow the server to compute or retrieve personalized ad recommendations without accessing the user’s profile. However, this also means that the server cannot directly observe which ad was shown or clicked. This limits traditional tracking and conversion.

*“Why not encrypt everything?”* If all data is encrypted, it would be necessary for servers to train directly on encrypted data, meaning that network performance is not readily available to the server (i.e., even the loss value is encrypted so observing training dynamics is not possible).

## 9. Related Work

We review related work on private inference, embedding compression techniques, and FHE compilers, with a focus on approaches relevant to encrypted inference in large-scale machine learning models, such as recommendation systems.

### 9.1. Private Inference

In a client-server model, private inference protects the model weights from clients, while also ensuring that the server does not learn any information from the client’s private data. Private inference has been extensively explored in recent years, mainly from a cryptographic perspective using methods such as homomorphic encryption or multiparty computation. In particular, private inference protocols relying on homomorphic encryption process the entire network on the server side and replaces the nonlinear layers with either low-degree polynomials [41] or high-degree composite approximations [42]. The former replacement strategy reduces the computational overhead at the cost of a potential accuracy loss.

While early works such as CryptoNets [30], MiniONN [43], GAZELLE [44], or DiNN [45] demonstrated the feasibility of evaluating neural networks on encrypted data, subsequent efforts have adapted homomorphic encryption for more complex tasks, including natural language processing and recommendation systems. In the context of NLP, a central challenge lies in the private evaluation of embedding layers, which typically involve large token dictionaries and table lookups.

Focusing on CKKS-based solutions, prior approaches have avoided large-sized LUT evaluations in different ways, by either sending high-dimensional encrypted one-hot vectors to the server [46], or by performing the embedding layer in plaintext on the client side [47], [48], under the assumption that the client has access to the embedding model. However, both methods have limitations, particularly when embedding tables are proprietary and bandwidth efficiency is critical. To address this, [1] proposed a solution that enables efficient encrypted evaluation of large LUTs within the CKKS framework along with some compression techniques which we discuss in the next section.

## 9.2. Compressing Embedding Tables

Embedding tables are a fundamental component in deep learning recommendation models that process categorical inputs with very large vocabularies. However, these tables can grow extremely large, consuming significant memory and bandwidth and becoming a bottleneck for training and inference. To address these challenges, many embedding compression techniques have been proposed to reduce storage and computational overhead while preserving the model's performance. These methods are broadly categorized as follows.

**Quantization-based embedding compression.** Quantization techniques reduce the size and computational cost of embedding tables by either converting floating-point values in embedding tables into lower-precision data types or by using clustering-based methods to share representations across similar embeddings. The former approach includes methods like uniform or row-wise quantization, which directly reduce memory usage by limiting numerical precision. Quantization methods can either be applied during training (quantization-aware training, QAT) [49], [50], [51] or post training [52], [53].

Clustering-based quantization techniques, such as product quantization or residual vector quantization, represent embeddings using a small number of learned centroids. These centroids are stored in compact data structures called codebooks, and each embedding vector is approximated by selecting and combining one or more codewords from these codebooks. This approach, often referred to as codebook-based compression, enables efficient storage and fast reconstruction of embeddings. It is leveraged in works like [36], where embeddings are expressed as compositions of shared codebook entries to achieve high compression rates.

**Hashing-based embedding compression.** One of the most widely used hashing-based strategies for compressing embedding tables is the Hashing Trick [54], which maps large input IDs into a smaller embedding space using a hash function. This reduces memory requirements by allowing multiple inputs to share the same embedding slot. However, this comes at the cost of potential collisions where unrelated IDs may be mapped to the same representation. This can affect the model's ability to differentiate between distinct inputs and introduce semantic overlap that reduces the model's overall performance. To reduce hash collision, Facebook introduced the Quotient-Remainder trick [37]. Hashing-based techniques are used in [55], [56] for example often in combination with other aforementioned methods.

**Adaptive compression methods.** Adaptive compression methods dynamically allocate memory based on feature importance. Instead of applying a fixed compression strategy uniformly across all features, these methods focus more on frequently accessed or high-impact features, while compressing infrequent ones a lot less. One example is CAFE+ [39] which uses a feature monitor to continuously keep track of the importance of each feature. CAFE+ also combines this dynamic strategy with hash-based and quantized representations to achieve high compression rates in extremely large-scale DLRMs.

**Decomposition methods.** Decomposition-based approaches such as Tucker decomposition, Tensor-Train or CP, reduce embedding size by representing high-dimensional tables using structured low-rank formats. In the context of DLRMs, [57] applies tensor-train decomposition to express large embedding tables as a sequence of smaller matrix products.

## 9.3. FHE Compilers

Compiler toolchains for FHE can be broadly categorized into circuit-level and domain-specific compilers. Circuit-level tools [11], [15], [16], [58], [59] focus on low-level optimization and scheduling for general-purpose programs. While they provide fine-grained control over the cryptographic operations, they generally target small workloads and lack support for complex machine learning tasks.

On the other hand, domain-specific compilers are designed to support machine learning workloads by abstracting cryptographic details and providing more high-level programming interfaces. Earlier works include CHET [13] and EVA [12], both implemented in the SEAL library [60]. Subsequent frameworks such as nGraph-HE [61], [62], TenSEAL [63], SEALion [64] and Concrete-ML [65] provide python-based interfaces to run encrypted inference for simple classifiers or quantized networks. More recently compilers such as Dacapo [66], HeLayers [67], Phelipe [14] and Orion [18] improve upon these later worksby introducing automating bootstrap placement and support for deeper models.

## 10. Conclusion

In this paper, we presented the first implementation of a private Deep Learning Recommendation Model (DLRM) using Fully Homomorphic Encryption (FHE), which we call HE-LRM. Unlike classical neural networks, DLRMs accept both dense and sparse features as input where sparse features are processed via large embedding lookup tables. To this end, we apply an FHE-friendly compression technique to embedding tables and improve upon existing methods with up to a 77× speedup. Furthermore, we present a multi-embedding packing strategy that enables parallel embedding table lookups in FHE. We apply HE-LRM to two datasets, UCI Heart Disease and Criteo Kaggle, demonstrating private inference in 24 seconds and 227 seconds respectively, while empolying our compression technique to the larger embedding tables of the Criteo dataset. We also exhibit training curves and performance trade-offs for compressed DLRMs showing that training loss curves and final test AUC are minimally impacted by our compression technique. These results demonstrate the feasibility of deploying FHE-based recommendation systems and provide the first end-to-end instantiation of a private recommendation model.

## Acknowledgements

This work was supported in part by Graduate Assistance in Areas of National Need (GAANN). The research was developed with funding from the NSF CAREER award #2340137 and DARPA, under the Data Protection in Virtual Environments (DPRIVE) program, contract HR0011-21-9-0003. Reagen and Ebel received a gift award from Google. The views, opinions, and/or findings expressed are those of the authors and do not necessarily reflect the views of sponsors.

## References

[1] J. yun Kim, S. Park, J. Lee, and J. H. Cheon, "Privacy-preserving embedding via look-up table evaluation with fully homomorphic encryption," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024, available at https://openreview.net/forum?id=apxON2uH4N.

[2] M. Naumov, D. Mudigere, H.-J. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C.-J. Wu, A. G. Azzolini *et al.*, "Deep learning recommendation model for personalization and recommendation systems," *arXiv preprint arXiv:1906.00091*, 2019.

[3] J. H. Cheon, G. Hanrot, J. Kim, and D. Stehlé, "Ship: A shallow and highly parallelizable ckks bootstrapping algorithm," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2025, pp. 398–428.

[4] J.-S. Coron and T. Seuré, "Paco: Bootstrapping for ckks via partial coefftoslot," *Cryptology ePrint Archive*, 2025.

[5] I. Chillotti, D. Ligier, J.-B. Orfila, and S. Tap, "Improved programmable bootstrapping with larger precision and efficient arithmetic circuits for tfhe," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2021, pp. 670–699.

[6] C. Lee, S. Min, J. Seo, and Y. Song, "Faster tfhe bootstrapping with block binary keys," in *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, 2023, pp. 2–13.

[7] J. Kim, J. Seo, and Y. Song, "Simpler and faster bfv bootstrapping for arbitrary plaintext modulus from ckks," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 2535–2546.

[8] B. Xiang, J. Zhang, Y. Deng, Y. Dai, and D. Feng, "Fast blind rotation for bootstrapping fhes," in *Annual International Cryptology Conference*. Springer, 2023, pp. 3–36.

[9] Z. Wang and M. Ikeda, "High-throughput key switching accelerator for homomorphic encryption," in *2023 International Conference on IC Design and Technology (ICICDT)*. IEEE, 2023, pp. 100–103.

[10] J. Mono and T. Güneysu, "A new perspective on key switching for bgv-like schemes," *Cryptology ePrint Archive*, 2023.

[11] M. Cowan, D. Dangwal, A. Alaghi, C. Trippel, V. T. Lee, and B. Reagen, "Porcupine: A synthesizing compiler for vectorized homomorphic encryption," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 375–389.

[12] R. Dathathri, B. Kostova, O. Saarikivi, W. Dai, K. Laine, and M. Musuvathi, "Eva: An encrypted vector arithmetic language and compiler for efficient homomorphic computation," in *Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation*, 2020, pp. 546–561.

[13] R. Dathathri, O. Saarikivi, H. Chen, K. Laine, K. Lauter, S. Maleki, M. Musuvathi, and T. Mytkowicz, "Chet: an optimizing compiler for fully-homomorphic neural-network inferencing," in *Proceedings of the 40th ACM SIGPLAN conference on programming language design and implementation*, 2019, pp. 142–156.

[14] A. Krastev, N. Samardzic, S. Langowski, S. Devadas, and D. Sanchez, "A tensor compiler with automatic data packing for simple and efficient fully homomorphic encryption," *Proceedings of the ACM on Programming Languages*, vol. 8, no. PLDI, pp. 126–150, 2024.

[15] R. Malik, K. Sheth, and M. Kulkarni, "Coyote: A compiler for vectorizing encrypted arithmetic circuits," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023, pp. 118–133.

[16] A. Viand, P. Jattke, M. Haller, and A. Hithnawi, "Heco: fully homomorphic encryption compiler," in *Proceedings of the 32nd USENIX Conference on Security Symposium*, ser. SEC '23. USA: USENIX Association, 2023.

[17] N. Neda, A. Ebel, B. Reynwar, and B. Reagen, " CiFlow: Dataflow Analysis and Optimization of Key Switching for Homomorphic Encryption ," in *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2024, pp. 61–72. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ISPASS61541.2024.00016

[18] A. Ebel, K. Garimella, and B. Reagen, "Orion: A fully homomorphic encryption framework for deep learning," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 734–749. [Online]. Available: https://doi.org/10.1145/3676641.3716008

[19] A. Ebel and B. Reagen, "Osiris: A systolic approach to accelerating fully homomorphic encryption," *arXiv preprint arXiv:2408.09593*, 2024.

[20] N. Samardzic, A. Feldmann, A. Krastev, S. Devadas, R. Dreslinski, C. Peikert, and D. Sanchez, "F1: A fast and programmable accelerator for fully homomorphic encryption," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 238–252.

[21] J. Kim, S. Kim, J. Choi, J. Park, D. Kim, and J. H. Ahn, "Sharp: A short-word hierarchical accelerator for robust and practical fully homomorphic encryption," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–15.

[22] B. Reagen, W.-S. Choi, Y. Ko, V. T. Lee, H.-H. S. Lee, G.-Y. Wei, and D. Brooks, "Cheetah: Optimizing and accelerating homomorphic encryption for private inference," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 26–39.

[23] N. Samardzic, A. Feldmann, A. Krastev, N. Manohar, N. Genise, S. Devadas, K. Eldefrawy, C. Peikert, and D. Sanchez, "Craterlake: a hardware accelerator for efficient unbounded computation on encrypted data," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 173–187.

[24] D. Soni, N. Neda, N. Zhang, B. Reynwar, H. Gamil, B. Heyman, M. Nabeel, A. Al Badawi, Y. Polyakov, K. Canida *et al.*, "Rpu: The ring processing unit," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2023, pp. 272–282.

[25] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "A full RNS variant of approximate homomorphic encryption," Cryptology ePrint Archive, Paper 2018/931, 2018. [Online]. Available: https://eprint.iacr.org/2018/931

[26] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Advances in cryptology–ASIACRYPT 2017: 23rd international conference on the theory and applications of cryptology and information security, Hong kong, China, December 3-7, 2017, proceedings, part i 23*. Springer, 2017, pp. 409–437.

[27] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song, "Bootstrapping for approximate homomorphic encryption," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2018, pp. 360–384.

[28] J.-P. Bossuat, C. Mouchet, J. Troncoso-Pastoriza, and J.-P. Hubaux, "Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys," Cryptology ePrint Archive, Paper 2020/1203, 2020. [Online]. Available: https://eprint.iacr.org/2020/1203

[29] D. Evans, V. Kolesnikov, and M. Rosulek, "A pragmatic introduction to secure multi-party computation," *Found. Trends Priv. Secur.*, vol. 2, no. 2–3, p. 70–246, Dec. 2018. [Online]. Available: https://doi.org/10.1561/3300000019

[30] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy," in *International conference on machine learning*. PMLR, 2016, pp. 201–210.

[31] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, "GAZELLE: A low latency framework for secure neural network inference," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 1651–1669. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/juvekar

[32] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa, "Delphi: A cryptographic inference service for neural networks," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2505–2522. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/mishra

[33] D. Rathee, M. Rathee, N. Kumar, N. Chandran, D. Gupta, A. Rastogi, and R. Sharma, "Cryptflow2: Practical 2-party secure inference," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 325–342. [Online]. Available: https://doi.org/10.1145/3372297.3417274

[34] K. Garimella, Z. Ghodsi, N. K. Jha, S. Garg, and B. Reagen, "Characterizing and optimizing end-to-end systems for private inference," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 89–104. [Online]. Available: https://doi.org/10.1145/3582016.3582065

[35] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," 2019. [Online]. Available: https://arxiv.org/abs/1912.01703

[36] R. Shu and H. Nakayama, "Compressing word embeddings via deep compositional code learning," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2018, available at https://openreview.net/forum?id=BJRZzFlRb.

[37] H.-J. M. Shi, D. Mudigere, M. Naumov, and J. 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*, 2020, pp. 165–175.

[38] A. Janosi, W. Steinbrunn, M. Pfisterer, and R. Detrano, "Heart Disease," UCI Machine Learning Repository, 1989, DOI: https://doi.org/10.24432/C52P4X.

[39] Z. Liu, H. Zhang, B. Chen, Z. Jiang, Y. Zhao, Y. Tao, T. Yang, and B. Cui, "Cafe+: Towards compact, adaptive, and fast embedding for large-scale online recommendation models," *ACM Transactions on Information Systems*, 2025.

[40] H. Muhammad, "htop - an interactive process viewer for unix systems," 2004, accessed: 2025-06-12. [Online]. Available: https://htop.dev/

[41] K. Garimella, N. K. Jha, and B. Reagen, "Sisyphus: A cautionary tale of using low-degree polynomial activations in privacy-preserving deep learning," *arXiv preprint arXiv:2107.12342*, 2021.

[42] E. Lee, J.-W. Lee, J.-S. No, and Y.-S. Kim, "Minimax approximation of sign function by composite polynomial for homomorphic comparison," Cryptology ePrint Archive, Paper 2020/834, 2020. [Online]. Available: https://eprint.iacr.org/2020/834

[43] J. Liu, M. Juuti, Y. Lu, and N. Asokan, "Oblivious neural network predictions via minionn transformations," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 619–631.

[44] C. Juvekar, V. Vaikuntanathan, and A. Chandrakasan, "{GAZELLE}: A low latency framework for secure neural network inference," in *27th USENIX security symposium (USENIX security 18)*, 2018, pp. 1651–1669.

[45] F. Bourse, M. Minelli, M. Minihold, and P. Paillier, "Fast homomorphic evaluation of deep discretized neural networks," in *Advances in Cryptology–CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part III 38*. Springer, 2018, pp. 483–512.

[46] A. Al Badawi, L. Hoang, C. F. Mun, K. Laine, and K. M. M. Aung, "Privft: Private and fast text classification with homomorphic encryption," *IEEE Access*, vol. 8, pp. 226 544–226 556, 2020.

[47] G. Lee, M. Kim, J. H. Park, S.-w. Hwang, and J. H. Cheon, "Privacy-preserving text classification on bert embeddings with homomorphic encryption," *arXiv preprint arXiv:2210.02574*, 2022.

[48] T. Chen, H. Bao, S. Huang, L. Dong, B. Jiao, D. Jiang, H. Zhou, J. Li, and F. Wei, "The-x: Privacy-preserving transformer inference with homomorphic encryption," *arXiv preprint arXiv:2206.00216*, 2022.

[49] Y. Zhou, Z. Dong, E. Chan, D. Kalamkar, D. Marculescu, and K. Keutzer, "Dqrm: Deep quantized recommendation models," *arXiv preprint arXiv:2410.20046*, 2024.

[50] W.-C. Kang, D. Z. Cheng, T. Chen, X. Yi, D. Lin, L. Hong, and E. H. Chi, "Learning multi-granular quantized embeddings for large-vocab categorical features in recommender systems," in *Companion Proceedings of the Web Conference 2020*, 2020, pp. 562–566.

[51] Y. Ko, J.-S. Yu, H.-K. Bae, Y. Park, D. Lee, and S.-W. Kim, "Mascot: A quantization framework for efficient matrix factorization in recommender systems," in *2021 IEEE International Conference on Data Mining (ICDM)*. IEEE, 2021, pp. 290–299.

[52] H. Guan, A. Malevich, J. Yang, J. Park, and H. Yuen, "Post-training 4-bit quantization on embedding tables," *arXiv preprint arXiv:1911.02079*, 2019.

[53] Z. Deng, J. Park, P. T. P. Tang, H. Liu, J. Yang, H. Yuen, J. Huang, D. Khudia, X. Wei, E. Wen *et al.*, "Low-precision hardware architectures meet recommendation model inference at scale," *IEEE Micro*, vol. 41, no. 5, pp. 93–100, 2021.

[54] K. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg, "Feature hashing for large scale multitask learning," in *Proceedings of the 26th annual international conference on machine learning*, 2009, pp. 1113–1120.

[55] H. Tsang and T. Ahle, "Clustering the sketch: dynamic compression for embedding tables," *Advances in Neural Information Processing Systems*, vol. 36, pp. 72 155–72 180, 2023.

[56] B. Ghaemmaghami, M. Ozdal, R. Komuravelli, D. Korchev, D. Mudigere, K. Nair, and M. Naumov, "Learning to collide: Recommendation system model compression with learned hash functions," *arXiv preprint arXiv:2203.15837*, 2022.

[57] C. Yin, B. Acun, X. Liu, and C. Wu, "Tt-rec: Tensor train compression for deep learning recommendation model embeddings," *arXiv preprint arXiv: 2101.11714*, 2021.

[58] Y. Lee, S. Heo, S. Cheon, S. Jeong, C. Kim, E. Kim, D. Lee, and H. Kim, "Hecate: Performance-aware scale optimization for homomorphic encryption compiler," in *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2022, pp. 193–204.

[59] Y. Lee, S. Cheon, D. Kim, D. Lee, and H. Kim, "{ELASM}:{Error-Latency-Aware} scale management for fully homomorphic encryption," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4697–4714.

[60] "Microsoft SEAL (release 4.1)," https://github.com/Microsoft/SEAL, Jan. 2023, microsoft Research, Redmond, WA.

[61] F. Boemer, A. Costache, R. Cammarota, and C. Wierzynski, "ngraph-he2: A high-throughput framework for neural network inference on encrypted data," in *Proceedings of the 7th ACM workshop on encrypted computing & applied homomorphic cryptography*, 2019, pp. 45–56.

[62] F. Boemer, Y. Lao, R. Cammarota, and C. Wierzynski, "ngraph-he: a graph compiler for deep learning on homomorphically encrypted data," in *Proceedings of the 16th ACM international conference on computing frontiers*, 2019, pp. 3–13.

[63] A. Benaissa, B. Retiat, B. Cebere, and A. E. Belfedhal, "Tenseal: A library for encrypted tensor operations using homomorphic encryption. arxiv 2021," *arXiv preprint arXiv:2104.03152*, 2021.

[64] T. van Elsloo, G. Patrini, and H. Ivey-Law, "Sealion: A framework for neural network inference on encrypted data," *arXiv preprint arXiv:1904.12840*, 2019.

[65] Zama, "Concrete ML: a privacy-preserving machine learning library using fully homomorphic encryption for data scientists," 2022, https://github.com/zama-ai/concrete-ml.

[66] S. Cheon, Y. Lee, D. Kim, J. M. Lee, S. Jung, T. Kim, D. Lee, and H. Kim, "{DaCapo}: Automatic bootstrapping management for efficient fully homomorphic encryption," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 6993–7010.

[67] E. Aharoni, A. Adir, M. Baruch, N. Drucker, G. Ezov, A. Farkash, L. Greenberg, R. Masalha, G. Moshkowich, D. Murik *et al.*, "Helayers: A tile tensors framework for large neural networks on encrypted data," *arXiv preprint arXiv:2011.01805*, 2020.