LEARNING COMPRESSED EMBEDDINGS FOR ON-DEVICE INFERENCE

Niketan Pansare [1] **Jay Katukuri** [1 2 *] **Aditya Arora** [1 *] **Frank Cipollone** [1] **Riyaaz Shaik** [1] **Noyan Tokgozoglu** [3] **Chandru Venkataraman** [1]

ABSTRACT

In deep learning, embeddings are widely used to represent categorical entities such as words, apps, and movies. An embedding layer maps each entity to a unique vector, causing the layer's memory requirement to be proportional to the number of entities. In the recommendation domain, a given category can have hundreds of thousands of entities, and its embedding layer can take gigabytes of memory. The scale of these networks makes them difficult to deploy in resource constrained environments, such as smartphones. In this paper, we propose a novel approach for reducing the size of an embedding table while still mapping each entity to its own unique embedding. Rather than maintaining the full embedding table, we construct each entity's embedding "on the fly" using two separate embedding tables. The first table employs hashing to force multiple entities to share an embedding. The second table contains one trainable weight per entity, allowing the model to distinguish between entities sharing the same embedding. Since these two tables are trained jointly, the network is able to learn a unique embedding per entity, helping it maintain a discriminative capability similar to a model with an uncompressed embedding table. We call this approach MEmCom (Multi-Embedding Compression). We compare with state-of-the-art model compression techniques for multiple problem classes including classification and ranking using datasets from various domains. On four popular recommender system datasets, MEmCom had a 4% relative loss in nDCG while compressing the input embedding sizes of our recommendation models by 16x, 4x, 12x, and 40x. MEmCom outperforms the state-of-the-art model compression techniques, which achieved 16%, 6%, 10%, and 8% relative loss in nDCG at the respective compression ratios. Additionally, MEmCom is able to compress the RankNet ranking model by 32x on a dataset with millions of users' interactions with games while incurring only a 1% relative loss in nDCG.

1 INTRODUCTION

In many modern deep learning systems, categorical features with large vocabulary sizes are some of the most important predictive features. Natural language processing systems rely heavily on inputs such as words, sub-word tokens, and individual characters, which are frequently fed directly into the models as categorical features. Additionally, search and recommender systems have increasingly represented model inputs as categorical features, shifting away from the focus on featurizing these inputs using more traditional and use-case specific methods. For example, queries (Grbovic et al., 2015), documents (Le et al., 2018), metadata (Vasile et al., 2016), and users have all been treated as categorical features in modern search and recommendation systems. This strategy has proven successful, improving model performance and user experiences.

[*]Work was done at Apple. [1]Apple, Cupertino, CA, USA [2]JP Morgan Chase, New York, NY, USA [3]Apple, New York, NY, USA. Correspondence to: Niketan Pansare <niketan_pansare@apple.com>.

Proceedings of the 5^{th} MLSys Conference, Santa Clara, CA, USA, 2022.

Research into systems that rely on categorical features have focused heavily on the use of embeddings. Systems like Word2Vec by Mikolov et. al. (Mikolov et al., 2013) and fastText by Bojanowski et. al. (Bojanowski et al., 2017) are able to learn embeddings that can be used to represent each of the entities in the vocabulary of a given categorical feature. Systems like BERT by Devlin et. al. (Devlin et al., 2019) take this idea a step further and are able to generate embeddings "on the fly" using the entity and the entity's context. While these systems initially targeted natural language processing use cases, the same ideas have been applied broadly. Today, both pre-trained embeddings and embeddings trained from scratch have become common practice in constructing neural networks (Devlin et al., 2019; Peters et al., 2018). In fact, these core ideas have become so popular in deep learning that the original embedding papers have garnered tens of thousands of citations.

The utility of embeddings for dealing with categorical features and adoption of these techniques has accelerated. However, in certain cases, there are important engineering challenges that arise while leveraging embeddings. In particular, as the vocabulary sizes of categorical features grow and as the number of categorical features in a single model grow,

the size of the embedding matrices and memory footprint also grow. This problem is particularly salient for search and recommender systems. While natural language vocabularies are often comprised of tens of thousands of words, the vocabularies of queries, documents, and metadata can easily grow into the millions. The resulting embedding matrices are quite large, and some operations on these models can become prohibitive, especially in low resource settings such as smartphones. Large embedding layers can be an issue even when with server-side inference. For example, an embedding table for a large social network use-case can be on the order of tens of gigabytes and hence can impact the performance of server-side inference (Gupta et al., 2020). To alleviate this issue, the researchers at Facebook and Twitter proposed *quotient-remainder* trick and *double hashing* respectively to reduce the size of the embedding table (Shi et al., 2019; Zhang et al., 2020). In section 5, we show that our approach outperforms both these techniques.

The size of the embedding table is even more important if the recommendation model has multiple embedding tables of comparable sizes or if the inference is done on-device. On-device inference, especially for recommender systems, has three key advantages. First, it has low inference latency (milliseconds rather than tens of milliseconds) thanks to the specialized processors on modern smartphones that can perform on the order of trillions of operations per second. Second, it can be performed in a privacy-friendly manner without requiring user's potentially sensitive data be sent to the servers. Third, it reduces the cost of hosting the recommendation service. These considerations are especially important given the impact of recommender systems; 35% of Amazon's revenue, 23.7% BestBuy's growth, 75% Netflix's video consumption and 60% YouTube's views come from their recommendation system (Xie et al., 2018).

Overall, we make the following contributions:

- We propose a novel technique to compress embeddings with minimal loss of accuracy. In section 4, we highlight three properties satisfied by our technique that make it ideal for compressing the embeddings over multiple tasks and datasets, as opposed to other state-of-the-art techniques.
- We validate the utility of our approach by comparing it with the state-of-the-art techniques on two classes of problems, classification and ranking, on five public datasets (Newsgroup, MovieLens, Netflix, Million Songs and Google Local Review). As a bonus, we include the results on two large-scale industry datasets (Games and Arcade) that confirms the findings of the public datasets.
- Finally, we evaluate the CPU and memory impact of our technique on Apple iPhone 12 Pro and Google Pixel 2 using popular on-device frameworks, CoreML and TensorFlow Lite respectively.

2 Related Works

Here we review four broad approaches to reduce the size of the embedding layer. We can reduce the size of the embedding (section 2.2) or the number of embeddings (section 2.3). Additionally, we can reduce the floating point precision (section 2.4) and also consider sparsifying the embeddings (section 2.5). In this paper, we will primarily focus on the first two approaches. The latter two approaches can be implemented on top of the first two.

2.1 Embeddings

In general, our work on compression builds off of the wealth of research that has gone into embeddings over the past decade. This research began with Word2Vec by Mikolov et. al. (Mikolov et al., 2013), which is a method for assigning a static embedding to each word in a vocabulary such that similar words have similar embeddings. This method has since become a popular pre-processing step to allow pre-trained models to be leveraged for transfer learning from large corpora. An important step in embedding research came with contextual embeddings built by systems such as BERT (Devlin et al., 2019) and ELMo (Peters et al., 2018). These systems are able to incorporate the context in which a feature appears for each individual training or inference example. In general, contextual embeddings have been shown to outperform standard embeddings for a wide variety of use cases.

Importantly, while many systems rely on pre-trained contextual or non-contextual embeddings as a static pre-processing step for categorical features, many systems have increasingly allowed for the fine-tuning of embeddings or even for the training of embeddings from scratch to better fit their own use cases.

2.2 Low-rank approximation

The simplest approach to reduce the size of the embedding layer is to use lower embedding dimensions. Alternatively, we can decompose the embedding matrix $\mathbf{E}_{v,e}$ into two smaller matrices $\mathbf{U}_{v,h}, \mathbf{V}_{h,e}$ where $v =$ vocab size, $e =$ embedding size, $h =$ hidden size, and $h << e$. Lan et. al. (Lan et al., 2019) used this technique referred to as low-rank approximation, or factorized embedding parameterization (along with weight sharing), to reduce the number of parameters of BERT-large (Devlin et al., 2019) by 18x. Denil et. al. (Denil et al., 2013) showed that the neural networks are overparameterized, and it is possible to predict 95% of its parameters using this technique. Jaderberg et al. (Jaderberg et al., 2014) and Denton et al. (Denton et al., 2014) used

this technique to compress large convolutional networks for efficient inference with less than a 1% drop in accuracy.

2.3 Weight sharing

Since neural networks are overparameterized, it is possible to reduce the size of a layer via weight sharing. This paper proposes the sharing of embeddings for inputs in same hash bucket.

Shi et al. (Shi et al., 2019) proposed a similar idea, the *quotient-remainder* trick, for reducing the size of the embedding table. In this approach, the embedding table $\mathbf{E}_{v,e}$ is replaced by two embedding tables $\mathbf{U}_{m,e}$ and $\mathbf{V}_{v/m,\,e}$, and the embedding $\mathbf{E}_i$ for an item i is approximated using $\mathbf{U}_{i\ mod\ m,\,e} \odot \mathbf{V}_{k,e}$ here k is the quotient obtained by dividing i by m and $\odot$ denotes element-wise multiplication.

Instead of grouping the input features, HashedNets(Chen et al., 2015) groups the weights into much smaller hash buckets, i.e. the (virtual) weight $V_{i,j}$ is mapped to $w_{h(i,j)}$ where h is the hash function. The gradient with respect to the mapped weights can be computed using $\frac{\partial \mathcal{L}}{\partial w_k} = \sum_{i,j} \frac{\partial \mathcal{L}}{\partial V_{i,j}} \frac{\partial V_{i,j}}{\partial w_k}$. Gong et al. (Gong et al., 2014) proposed clustering the weights using k-means after training.

Using feature hashing for dimensionality reduction was first proposed by Weinberger et al. for large scale multitask learning (Weinberger et al., 2009). The authors showed that sparsity in the input features minimizes the hash collision and hence makes it more effective. To reduce the probability of collision even further, Zhang et al. proposed using two hash functions instead of one (Zhang et al., 2020).

2.4 Lower precision

Most deep learning frameworks use 32-bit floating point representations of parameters during training and the parameters are usually quantized before inference for storage and performance reasons. Since neural networks are shown to be resilient to noise (Murray & Edwards, 1994), lower precision has been proposed to reduce the size of the neural network as it can be modeled as noise (Han et al., 2016). Vanhoucke et al. (Vanhoucke et al., 2011) improved the performance of an early neural network for speech recognition using fixed-point (`int8`) instructions rather than 32-bit floating point instructions on CPUs without loss of accuracy. Krishnamoorthi (Krishnamoorthi, 2018) evaluated various techniques for quantizing convolutional neural networks with integer weights and activations. Zhu et al (Zhu et al., 2016) proposed reducing the precision of weights in neural networks to as low as two bits.

2.5 Sparsification

Han et al. (Han et al., 2015; 2016) focused on reducing the size (i.e. the number of non-zero parameters) of fully connected layers of AlexNet and VGG-16 networks using connection pruning, where the weights with small magnitudes are set to zero. Liu et al (Liu et al., 2015) reduced the size of convolutional layers of ResNet-152 network by 90% with just a 2% loss of accuracy. LeCun (Lecun et al., 1989) suggested using the second derivative of the objective function with respect to the parameters for pruning. These methods generate unstructured random connectivity and hence cannot exploit specialized sparse formats or sparse kernels for efficient execution. Structured pruning (Liu et al., 2018; Wen et al., 2016) addresses this issue by pruning a channel or a layer.

2.6 On-Device Inference

Compared to traditional computation paradigms, on-device inference provides several advantages that include improved latency, low communication bandwidth, and better data privacy. On-device inference with deep neural networks is challenging due to compute and memory resource constraints. These constraints are alleviated by methods discussed in recent research works in three directions: framework based, neural net architecture based, and hardware based optimisations.

We ran our performance experiment on iPhone and used a publicly available on-device inference framework called CoreML. CoreML contains direct support to convert models from frameworks like Caffe, TensorFlow, PyTorch to the CoreML format and perform inference using the CPU, GPU, and Neural Engine. Other popular framework for on-device inference are Caffe2 (Paszke et al., 2019), TensorFlow Lite (Abadi et al., 2015) and, Mobile AI Compute Engine.

Neural network architecture research has focused on optimising models for resource constraints on-device by techniques like pruning (Liu et al., 2018; Wen et al., 2016; Han et al., 2015; 2016; Lecun et al., 1989), quantization (Gong et al., 2014), weight sharing (Shi et al., 2019; Chen et al., 2015) , low rank approximations (Lan et al., 2019; Devlin et al., 2019; Denil et al., 2013; Jaderberg et al., 2014; Denton et al., 2014), and knowledge distillation (Hinton et al., 2015). These techniques reduce model size and achieve faster computation, with lower memory footprints, thereby reducing inference latency.

Hardware research involves making architectural changes for efficient on-device inference and publishing software development kits (SDKs) to expose the architecture improvements. However these enhancements are mostly vendor specific and cannot be reused across all platforms.

3 BACKGROUND

In neural networks, an embedding layer is typically used to map an input category to a feature vector. It does so by using an embedding matrix or table $\mathbf{E}^{v \times e}$ where v denotes the vocabulary size and e denotes the dimensions of the row vector, commonly referred to as the embedding size. There are two common ways to implement the embedding layer in a deep learning framework: matrix and table approach. In the matrix approach, we first map the input to a one-hot vector and then multiply it with the embedding matrix $\mathbf{E}$, whereas in the table approach, no such mapping is required and the embedding vector is obtained via a lookup operation.

The GPU operator for the embedding layer in Caffe[1] was initially implemented using the matrix approach. Almost all modern deep learning frameworks rely on the table approach for the reasons mentioned below. Since both approaches require us to store $\mathbf{E}$, the storage (on-disk memory) complexity for each approach is $O(v \times e)$. The runtime memory requirements for the matrix and table approaches are $O(v \times e + b \times (e + v))$ and $O(v \times e + b \times (e + 1))$ respectively, where b is the batch size. Expanding the batch size term shows that the final term in the matrix approach is $O(b \times v)$ as opposed to $O(b)$ in the table approach. In other words, as the vocabulary size increases, the table approach scales in terms of the dimension of the row vector, whereas the matrix approach scales in terms of both the dimension of the row vector and the batch size.

Assuming a vocabulary of 100k (which is common for a recommender system) and a batch size of 1 (i.e. smallest possible batch size), the memory required for the table approach for the embedding size 128 and 256 is 51 MB and 102 MB. This is acceptable for desktop and workstation CPUs and GPUs but not for low-memory phones due to limited disk and memory space. Shipping such a model reliably to millions of devices with slow network speeds is challenging, and it becomes even more difficult if the model is trained and updated frequently. To optimize the memory required for inference, on-device frameworks such as CoreML[2] and TensorFlow-Lite (Abadi et al., 2015) use memory-mapped IO (via `mmap`) rather than loading the entire embedding table into the memory. Typically, the read performance of `mmap` is reasonably fast, but writes are much slower. Thus, the inference time for a memory-mapped embedding layer is negligible, but training time (typically done via Federated Learning (McMahan et al., 2017)) is much slower. Nonetheless, the on-disk memory requirements remain the same and hence models with **large embedding layers (in hundreds of megabytes) become prohibitively expensive for on-device inference**.

[1] `https://git.io/JzV9u`

[2] `https://developer.apple.com/machine-learning/core-ml/`

4 OUR APPROACH

For a technique to be effective in compressing the embedding layer for NLP and recommender systems, we propose that the compression technique must satisfy three properties:

1. The compressed embedding layer should minimize the number of categories that share an embedding vector. An optimal compression technique has capacity to map every category to a **unique vector**.
2. If the technique employs composition, the operator should be **simple** enough so as not to overly constrain the search space. This allows the technique to work well on a variety of machine learning tasks and datasets.
3. It should be well-suited even when the categories are distributed non-uniformly. For example: commonly used categories, such as words, movies, and apps, are typically **power law** distributed.

Before we detail the applicability of above properties for the relevant state-of-the-art techniques, we summarize them in the below table:

	Unique Vector	Simple Op.	Power-law
Low-rank approximation	Yes	N/A	No
Quotient-remainder	Yes	No	Yes
Naive hashing	No	N/A	Yes
Double-hashing	No	Yes	Yes
Our approach	Yes	Yes	Yes

Since low-rank approximation techniques, such as factorized embedding parameterization, effectively factorize the embedding matrix, they satisfy the first property but ignore the distribution of categories. As a result, these may be less-suited for compressing embedding layers when the categories are distributed according to power-law. In our experiments, factorized embedding parameterization performed poorly on all tasks and datasets, except Newsgroup, compared to other techniques. Similarly, the quotient-remainder trick implements QR decomposition of the embedding matrix, but unlike the low-rank approximation technique, it can handle category skew. However, we observed that it did not perform well in our experiments. We argue that the compositional operator (either concatenation or multiplication of quotient-remainder embeddings) is relatively complex to generalize over the datasets we tested.

Naive hashing does not guarantee a unique embedding vector for the given category and has a collision rate of $\frac{v}{m} - 1 + (1 - \frac{1}{m})^v$. To reduce the collision rate, Zhang et al. (Zhang et al., 2020) proposed double hashing, which has much lower collision rate of $\frac{v}{m^2} - 1 + (1 - \frac{1}{m^2})^v$. Nonetheless, it still cannot guarantee a unique embedding vector

for the given category. We propose Multi-Embedding Compression, or MEmCom for short, that allows for a unique vector per category and can handle power-law distributed categories.

Before we discuss MEmCom, let's review the *quotient-remainder* method (described by Algorithm 1) proposed by (Shi et al., 2019). The quotient operator is denoted using $\backslash$ and the element-wise multiplication operator is denoted using $\odot$.

Algorithm 1 Quotient-Remainder Trick (Shi et al., 2019)

Inputs: Input category x, Embedding tables $\mathbf{U} \in \Re^{m \times e}$ and $\mathbf{V} \in \Re^{\frac{v}{m} \times e}$
Output: Embedding vector associated with x
Determine index i of category x
Compute hash indices $j = i \bmod m$ and $k = i \backslash m$
Lookup embeddings $\mathbf{x}_{rem} = \mathbf{U}_j$ and $\mathbf{x}_{quo} = \mathbf{V}_k$
Return $\mathbf{x}_{rem} \odot \mathbf{x}_{quo}$

Note that the embedding size e is the same for the embedding tables $\mathbf{U}$ and $\mathbf{V}$, but the number of embeddings might be different (i.e. m and $\frac{v}{m}$) respectively. As such, the Quotient-Remainder Method reduces the memory complexity of the embedding layer from $O(v \times e)$ to $O\left((m + \frac{v}{m}) \times e\right)$.

Logically, we can think of the embedding layer as a set of v functions where $f_v(i) = \mathbf{E}_i$. The hashing method reduces the number of functions to m, which can be significantly smaller than that in the uncompressed model (i.e. v). To remedy this, Shi et al. (Shi et al., 2019) proposed combining the hashed embeddings (i.e. $\mathbf{x}_{rem}$) with the quotient embeddings (i.e. $\mathbf{x}_{quo}$) to produce to the final embeddings. With this method, the number of functions ($m \times \frac{v}{m}$) are same as that of the uncompressed model (i.e. v), but these functions are constrained rather than arbitrary functions.

Unlike quotient-remainder and double hashing, the embedding tables in MEmCom have hybrid shapes and require *broadcasting*. Broadcasting is an algorithmic technique that allows matrix/tensor frameworks such NumPy, TensorFlow, and PyTorch to handle arithmetic operations (elementwise multiplication in our case) of different shapes (Harris et al., 2020; Abadi et al., 2015; Paszke et al., 2019; Iverson, 1980). For example: if we multiply two matrices **A, B** of shape 3 X 4 and 3 X 1 respectively, we can loop over each column of **A** and multiply it with **B**. As looping is typically an expensive operation in these frameworks, broadcasted operators typically have low-level efficient implementation that avoids unnecessary copies of the data.

MEmCom employs a judicious design of the hybrid embedding tables along with easily generalizable linear relationship and a ubiquitous broadcasting operator to achieve a high compression ratio and low loss of overall accuracy compared to an uncompressed model. Instead of learning an e-dimensional embedding table for quotient, we learn a one-dimensional embedding table which can then be composed with the embeddings learned using the hashing method. Doing so guarantees that we learn v distinct functions $f_v(i) = \mathbf{E}_i$. We detail this technique in Algorithm 2. Since the embeddings **U** and **V** are learned jointly, the embeddings **U** learned using this technique and that by Algorithm 1 are likely going to be different. Using experimental evaluation, we show that this technique works well over different learning problems.

Algorithm 2 MEmCom (no bias)

Inputs: Input category x, Embedding tables $\mathbf{U} \in \Re^{m \times e}$ and $\mathbf{V} \in \Re^{v \times 1}$
Output: Embedding vector associated with x
Determine index i of category x (sorted by frequency)
Compute hash index $j = i \bmod m$
Lookup embeddings $\mathbf{x}_{rem} = \mathbf{U}_j$ and $\mathbf{x}_{mult} = \mathbf{V}_i$
Return $\mathbf{x}_{rem} \odot \mathbf{x}_{mult}$

We now extend the above algorithm to support bias. Unlike the quotient-remainder trick (i.e. Algorithm 1), $\mathbf{x}_{mult}$ and $\mathbf{x}_{bias}$ are broadcasted in our approach (i.e. Algorithm 2 and 3). Though adding $\mathbf{x}_{bias}$ allows MEmCom to generalize well on wider variety of tasks and datasets, in practice, MEmCom with no bias performs equally well.

Algorithm 3 MEmCom (with bias)

Inputs: Input category x, Embedding tables $\mathbf{U} \in \Re^{m \times e}$, $\mathbf{V} \in \Re^{v \times 1}$ and $\mathbf{W} \in \Re^{v \times 1}$
Output: Embedding vector associated with x
Determine index i of category x (sorted by frequency)
Compute hash index $j = i \bmod m$
Lookup embeddings $\mathbf{x}_{rem} = \mathbf{U}_j$, $\mathbf{x}_{mult} = \mathbf{V}_i$ and $\mathbf{x}_{bias} = \mathbf{W}_i$
Return $\mathbf{x}_{rem} \odot \mathbf{x}_{mult} + \mathbf{x}_{bias}$

5 Experimental Evaluation

Overview. In this section, we focus on three aspects of the problem that are relevant to on-device inference, namely (1) maximum compression of a given model, (2) minimal loss in accuracy, and (3) efficient on-device performance. By compression, we refer to reduction in the number of model parameters and hence the on-disk model size rather than in-memory model size. Since these models are typically sent to millions of devices (smartphones) over limited bandwidth, the on-disk model size is important. That said, it is possible that a small model on-disk can potentially have a high inference overhead, i.e. it might require a large amount of RAM or take longer to perform inference (such as Weinberger's hashing method (Weinberger et al., 2009)). To explore these aspects, we have two sets of experiments: one where we

	Section 5.2	Section 5.1	Section 5.3
Criteria	Memory (on-disk) v/s accuracy tradeoff		Inference time and memory footprint
Evaluate	Different model compression techniques		MEmComp
Baseline	Uncompressed model		Weinberger's hashing trick
Task	Ranking	Classification	
Model	Point-wise and Pair-wise Ranking Networks	Embedding-based Fully connected Feed-forward Network (see code 1)	
Dataset (see table 2)	MovieLens, Netflix, MSD, Google, Arcade	Games, Arcade, News-group	MovieLens, Netflix, MSD, Google, Games, Arcade
Results	Figure 2 and 3	Figure 1	Table 3

Table 1: Summary of Experimental Evaluation

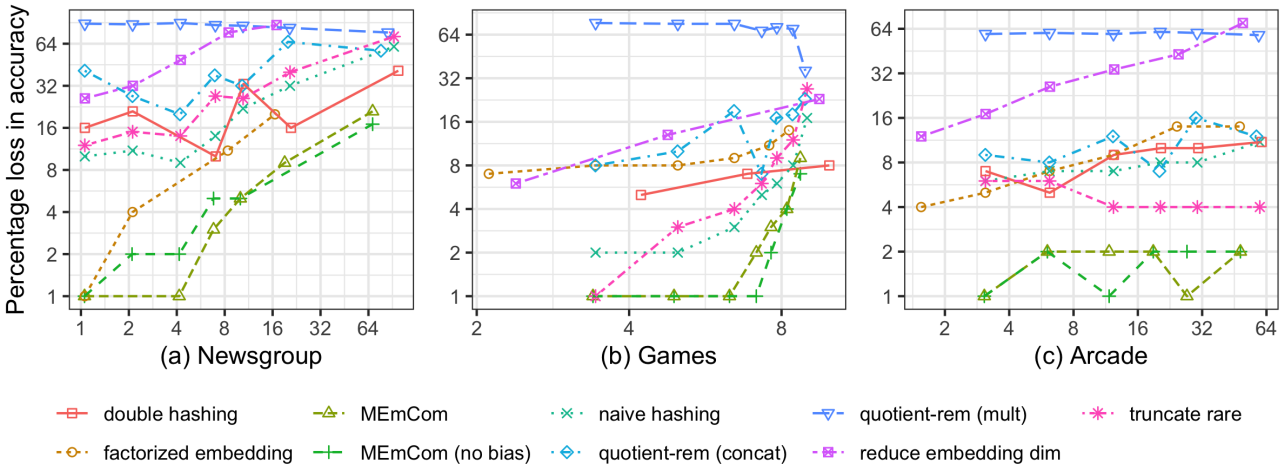

Figure 1: Evaluating compression vs accuracy tradeoff (classification). The x-axis shows the compression ratio.

evaluate the tradeoff between model size and model accuracy (see section 5.1 and 5.2)) and another where we focus on the inference overhead of `MEmCom` (our approach) (see section 5.3). For the first set of experiments, we compare our technique to the state-of-the art techniques to compress embeddings for two tasks: classification and ranking. These are discussed in sections 5.1 and 5.2 respectively.

Model. To be consistent across the experiments, we use a common network structure and hyper-parameters: an embedding-based, fully connected, feed-forward network. The network structure is a common strategy for learning latent user embeddings based on items that the user has interacted with in some way. For example, Nazari et al. (Nazari et al., 2020) used a similar architecture to perform podcast recommendations. Importantly, we observed optimal model performance using this network structure, having experimented with a variety of popular network structures and hyperparameters. The code 1 shows the Keras implementation of the baseline network for the classification experiment (section 5.1) where the embedding size is 256, the input length is 128, and the number of embeddings is equal to the size of the input vocabulary `V`.

```
embed = Embedding(input_dim=V, output_dim=256,
↪  input_length=128, mask_zero=False)(input)
l = AveragePooling1D(pool_size=128)(embed)
l = Flatten()(l)
l = ReLU()(l)
l = Dropout(dropout)(l)
l = BatchNormalization()(l)
l = Dense(units=int(embedding_size / 2),
↪  activation='relu')(l)
l = Dropout(dropout)(l)
l = BatchNormalization()(l)
output_layer = Dense(units=num_labels,
↪  activation='softmax')(l)
```

Code 1: Embedding-based Fully connected Feed-forward Network

Except for the embedding layer (line 1), we use the same network for all the techniques in this experiment. For the ranking experiments (section 5.2), we train the pointwise network and then use the softmax scores as the basis for ranking. We also alter the network by removing the Dense

layer following the Average Pooling, as this empirically proved to give us better performance. Additionally, we experiment with a pairwise siamese network where the above network is shared for the inputs. We explain this in more detail in the section 5.2.

State-of-the-art techniques. As described in section 2, we can group the techniques to compress embeddings into two categories, one that reduces the number of embeddings and one that reduces the embedding dimension. We first discuss the techniques that fall into the first category. (1) Naive hashing performs the `mod` operation on the input feature vector before the embedding table lookup. (2) In double hashing, we use two hash functions (and embedding layers) instead of one and concatenate their embeddings (Zhang et al., 2020). (3) We use two variants of the *quotient-remainder* method (Shi et al., 2019) described in the section 2.3, one where the compositional operator is *concatenation* and other where the compositional operator is *elementwise multiplication*. (4) Also, we compare these with algorithm 2 ("*MEmCom (our approach with no bias)*") and (5) algorithm 3 ("*MEmCom* (our approach)"). To validate that the hashing-based compression techniques yield useful results, we include a simple baseline where we drop the less popular apps ("*truncate rare*"). There are two techniques we evaluate that reduce the embedding dimension, namely *factorized embedding parameterization* (Lan et al., 2019) and by simply *reducing the number of embedding dimensions*. (6) For all methods except "reduce embedding dim," the output embedding dimensions is 256. For "reduce embedding dim," we progressively reduce the embedding dimensions by a factor of 2 (i.e. 128, 64, 32, 16, 8 and 4) for compressing the model. Each point on the below plots indicates a model trained using one of these hyperparameters. (7) For "factorized embedding," we keep the output embedding dimension the same (i.e. the number of hidden units of dense layer = 256), but vary the dimension of the embedding layer by a factor of 2 starting from 128. For all other approaches, we vary the number of embeddings using the mod m to vocabulary size, 100K, 50K, 25K, 10K, 5K and 1K. The results for `TT-Rec` (Yin et al., 2021) were similar to "factorized embedding" for all datasets; likely because both these approaches have large number of shared parameters, which in turn decreases the representational capacity of the embeddings. *Mixed dimension embeddings* (Ginart et al., 2019) is a blocked extension of "factorized embedding" with two additional hyperparameters, i.e., the number of blocks and the temperature (in case of popularity-based dimension sizing). In line with author's suggested rule of thumb, we set the number of blocks to the number of distinct categorical features, which in our case is 1. With this hyperparameter setting, the results were similar to that of the "factorized embedding" approach.

Datasets. For all the datasets, the input is a fixed length vector of size 128. (1) The "Newsgroup" dataset is fetched using scikit-learn's dataset API `sklearn.datasets.fetch_20newsgroups`. (2) The "MovieLens Ratings" (Harper & Konstan, 2015) dataset is a popular dataset consisting of 25M user ratings of movies, often used for benchmarking recommender systems. (3) The "Million Songs" (Bertin-Mahieux et al., 2011) dataset contains around 1M triplets of the form *user, song, number of listens*. This dataset is commonly used for benchmarking recommender systems as well. (4) The "Google Local Reviews" (He et al., 2017) dataset contains around 11M reviews of local businesses, along with locations and other metadata. (5) The "Netflix Ratings" (Bennett et al., 2009) dataset refers to the movie ratings dataset released by Netflix for their recommender system competition. In addition to the above publicly available datasets, we include two proprietary datasets: *Games* and *Arcade*, a random sample of mobile gaming platform users. The Games dataset is about 10x larger than the Arcade dataset and has a much larger output vocabulary. Even though these two datasets are not publicly available, we decided to include the corresponding experiments as they closely reflect the scale and complexity (especially in terms of behavioral signals) of a real-life large-scale recommendation dataset. The key differences between the above datasets are the lengths of input/output vocabularies and the distributions of the vocabularies (especially the output vocabularies). Each output vocabulary indirectly affects the number of parameters in the last layer of (see line 10 in the above code) and hence the size of each model. The different distribution of each output vocabulary provides diverse workloads for this experiment. We summarize the statistics of these datasets in the table 2.

Setup. We train our model using TensorFlow 1.12.3 and Keras 2.2.4 on a machine with a Nvidia V100 GPU and CUDA 9.0 and CuDNN 7.6.2.

5.1 Experiment 1: Compression vs. Accuracy tradeoff (Classification)

Experimental Setup. For each technique, we evaluate the loss of accuracy (compared to the baseline uncompressed network) and the compression ratio (size of baseline network / size of the compressed network) over three datasets: (1) 20 Newsgroups, (2) Games, and (3) Arcade. For all three datasets, the input vocabulary is of size 100K or greater. For the last two datasets, we use the previous 127 apps that the user purchased along with the user's country to predict the last app the user purchased. To simplify the setup, we use a shared vocabulary for the app identifiers and the countries. For example, if there are n countries and m apps, then the vocabulary is of size $n + m + 1$. The countries are mapped to ids 1 to n and the apps are mapped to ids $n + 1$ to $n + m$. The id 0 is reserved for padding. We used frequency-based mapping for the vocabulary, i.e. the most downloaded app

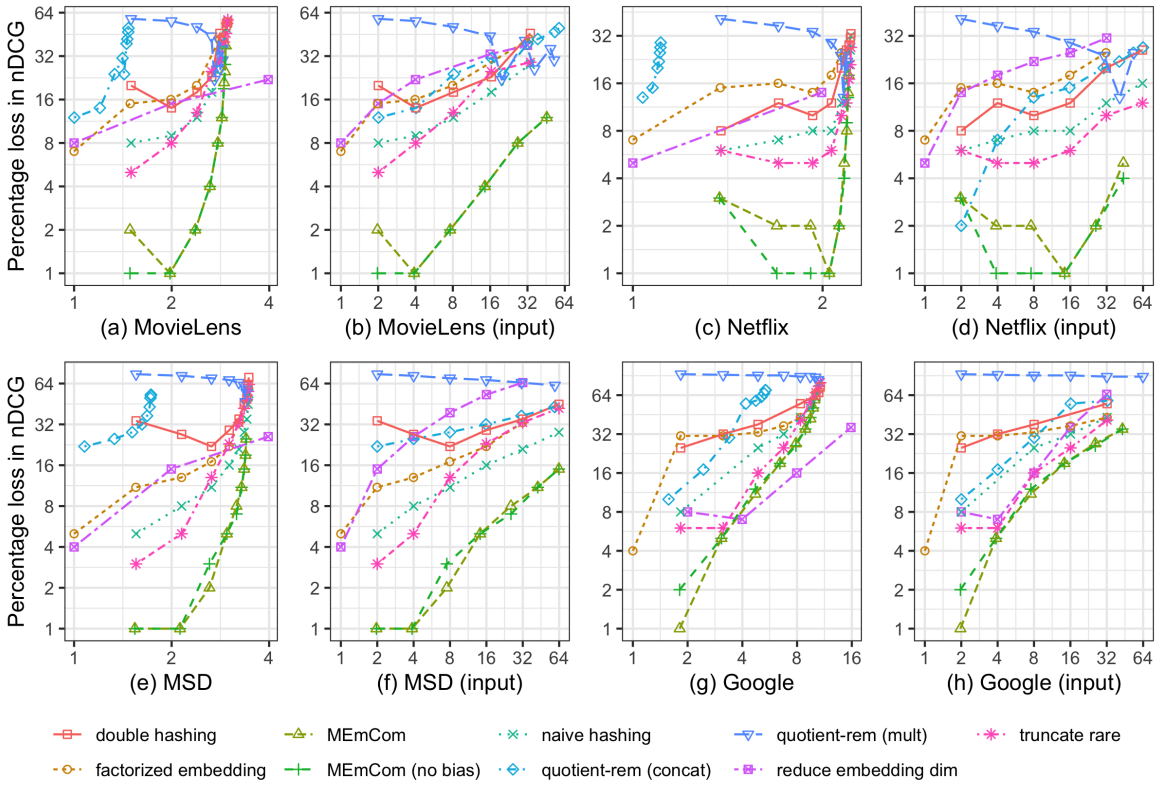

Figure 2: Evaluating compression vs accuracy tradeoff (Pointwise Ranking). The x-axis shows the compression ratio.

	Newsgroup	MovieLens	Million Songs	Google Local Reviews	Netflix	Games	Arcade
Number of training samples	11.3K	655K	4.5M	246K	2.1M	78M	7.5M
Number of evaluation samples	7.5K	72.8K	500K	27K	235K	65K	65K
Input vocabulary size	105K	10K	50K	200K	17K	480K	300K
Output vocabulary size	20	5K	20K	20K	16K	119K	145

Table 2: Datasets Used.

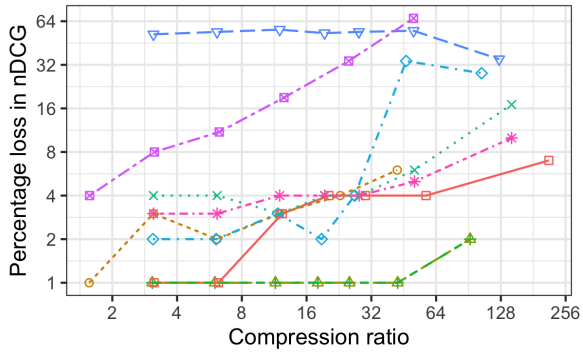

Figure 3: Evaluating compression vs accuracy tradeoff (Arcade - Pairwise Ranking). This figure has the same legend as the figure 2 .

is assigned the id $n+1$ and the country with most purchases is assigned the id 1. To ensure a fixed length feature vector, we drop the least recently purchased apps if the user has more than 127 apps and pad (with id 0) if the user has less than 127 purchases.

Results. Figure 1 shows the state-of-the-art techniques we explored for reducing the size of the embedding layer for Newsgroup, Games, and Arcade datasets. The x-axis shows the compression ratio, which is the ratio of the number of parameters of the uncompressed model and that of the technique for a given set of hyperparameters. For consistency across the datasets, we measure the number of parameters of all the layers and not just the embedding layers. Thus, the compression ratio of 2 implies that the proposed technique was able to compress the on-disk model size by half, but the embedding layer was compressed even further. The

y-axis shows the percentage loss in accuracy for the given technique compared to the uncompressed model. For all compression ratios, MEmCom has much lower loss in accuracy compared to other techniques. Other than MEmCom and factorized embedding parameterization, all other techniques did not work well for the Newsgroup dataset. On the Arcade dataset, a dumb compression technique "truncate rare" where we drop the rare apps, worked pretty well compared to more sophisticated, state-of-art compression techniques. Even in this case, MEmCom outperformed it by 2x.

5.2 Experiment 2: Compression vs nDCG tradeoff (Ranking)

Experimental Setup. For the ranking experiments, we worked with five datasets: (1) MovieLens Ratings (Harper & Konstan, 2015), (2) Million Songs (Bertin-Mahieux et al., 2011), (3) Google Local Reviews (He et al., 2017), (4) Netflix Ratings (Bennett et al., 2009), and (5) Arcade. While processing the Movielens Ratings, Google Local Reviews, Million Songs, and Netflix Ratings datasets, we produce up to five training examples per user. The training label for each example will come from the set of the most recent item interactions, with the 128 most recent item interactions (excepting the label) serving as the training input data. Additionally, we filtered items that did not have sufficient popularity in the training data. This decision allows sufficient data to train embeddings that perform well and incents the models to be as efficient as possible with the allowed number of parameters, allowing us to more easily compare different compression techniques. The table in 5.1 provides information on the number of items allowed in the input and the output of the models for each dataset, For the network architecture, we chose to use the same architecture described by the Keras code in the introduction to this section, setting up a pointwise learning-to-rank network. We altered the network by removing the Dense layer following the Average Pooling, as this proved empirically to give us better performance.

Due to our training data design, we use the softmax as our loss function as in the classification experiments. After training, we predict on the evaluation data using the softmax scores to compute the final ranking score for each item. This simplified architecture allows us to focus on the impact of the compression techniques we are exploring. For the Arcade dataset, instead of using a pointwise ranking model, we use a pairwise ranking model with the *RankNet* (Burges et al., 2005) architecture (for greater coverage of different types of models in the experiments). This network takes as input user features and two item IDs such that the first item is ranked higher than the second item. It outputs two scores corresponding to the input item ids, and during training, we maximize the difference between these scores. If the network is able to differentiate between arbitrary pairs of item ids in the evaluation dataset, it has learned the ranking function and then can be used to rank any list of items available in the output vocabulary. We use the Arcade dataset described earlier and use this network to rank the Arcade games. We evaluate a popular ranking metric, normalized discounted cumulative gain (or nDCG for short) (Valizadegan et al., 2009), for evaluation.

Results. Figures 2 and 3 show the percentage loss in nDCG for the techniques we evaluated for reducing the embedding layer compared to an uncompressed model. Figure 2 shows the results of a pointwise learning-to-rank network on the above mentioned dataset, while figure 3 shows the results of the pairwise ranking model on the Arcade dataset. Like earlier experiments, we prefer techniques that have minimal loss in nDCG for a given compression ratio. As shown in figure 3, MEmCom has less than 1% loss in nDCG while compressing the Arcade ranking model by 32x. Additionally, we achieved an approximately 4% loss in nDCG while compressing the input embedding matrices of the Movielens Ratings, Google Local Reviews, Million Songs, and Netflix Ratings models by 16x, 4x, 12x, and 40x, respectively, beating out other state-of-the-art model compression techniques. These results are shown in figures 2 (a), 2 (c), 2 (e) and 2 (g), along with the compression ratios of the full models. As MEmCom with and without bias performs exactly the same, their lines overlap in the figure3.

5.3 Experiment 3: Runtime Performance of On-Device Inference

Experimental Setup. In this experiment we compare the runtime performance of a model that uses our approach described in the section 5.1 with a model that applies Weinberger's hashing method (Weinberger et al., 2009) on the one-hot encoded input features. To keep the comparison fair, both models have the same set of layers except for the first embedding layer and also the same fixed hash size of 10K is used in both models. Because the multi-embedding approach, the quotient trick and double hashing all rely on "hashing the number of embeddings," these results are applicable for those approaches too. The two models were benchmarked with the datasets discussed in the sections 5.1 and 5.2. The benchmarks were performed on two smartphones, an Apple iPhone 12 Pro and a Google Pixel 2, using the on-device frameworks, CoreML 4.1.4, and TensorFlow-Lite 2.3.0 (TF-Lite) respectively. We report the average values across 1000 benchmark runs. Since the smartphones are from different generations, the goal is not to compare their performance, nor the performance of the corresponding on-device frameworks. Instead, we focus on comparing MEmCom with Weinberger's hashing method for a given setup. CoreML allows a developer to constrain the possible set of computes using an enum MLComputeUnits, which can be set to all, cpuOnly or cpuAndGPU. But

		Inference Time				Memory footprint			
		CoreML			TF Lite	CoreML			TF Lite
		all	cpuOnly	cpuAndGPU	CPU	all	cpuOnly	cpuAndGPU	CPU
Newsgroup	MEmCom	0.21	0.38	0.47	0.18	3.23	2.88	4.88	1.55
	Weinberger	0.89	0.9	0.95	30.96	27.57	27.62	28.29	8.21
Movielens	MEmCom	0.07	0.06	0.13	0.05	2.6	2.8	4.56	1.04
	Weinberger	0.9	0.91	0.95	30.84	27.6	27.6	28.4	8.21
Million Songs	MEmCom	0.07	0.06	0.12	0.07	2.7	2.49	4.34	1.24
	Weinberger	0.91	0.9	0.96	30.6	27.8	27.9	28.5	8.22
Google Local Review	MEmCom	3.49	3.34	3.42	0.4	5.34	4.30	5.83	3.44
	Weinberger	1.19	1.2	1.25	30.91	10	31.7	32.6	9.25
Netflix	MEmCom	1.22	0.60	0.76	1.22	8.65	2.64	4.24	8.6
	Weinberger	1.32	1.32	1.42	31.4	10.6	37.8	38.5	31.4
Games	MEmCom	3.42	3.33	3.42	4.4	5.39	4.22	5.81	31.2
	Weinberger	2.51	2.53	2.64	34.6	13.2	16.2	16.2	37.5
Arcade	MEmCom	0.06	0.06	0.12	0.01	3.63	2.52	4.38	1.36
	Weinberger	1.14	1.15	1.18	30.9	10.2	29.1	30	37.6

Table 3: Inference time (in milliseconds) and memory footprint (in megabytes) on different devices (batch size = 1, FP32) comparing MEmCom (no bias) and Weinberger Hashing

with the exception of the CPU, the CoreML API does not allow the developer to force the execution of the model on a specific compute unit. For example, if the enum is set to `all`, CoreML can schedule an operator in the model (or the entire model) on the best compute unit available, which can be Neural Engine, CPU or GPU. TF-Lite's benchmarking tool was used to measure the performance of the corresponding models. This tool allows the measurement of execution on a given compute using the flag `use_gpu`. For GPU execution, TF-Lite tries to delegate ineligible operators (one-hot operator) on CPU and schedules the remaining on the GPU. However, our execution fails as one of the operators (`reduce_sum`) scheduled to be executed on GPU does not have a corresponding implementation. Hence, we do not include the results for GPU execution with TF-Lite. The models were not quantized during compilation; the parameters are stored in 32-bit precision and the computation is also performed in the same precision. After model initialization, the inference time for baseline uncompressed model is comparable to `MEmCom`. The on device training time will be much lower for `MEmCom` compared to non-compressed baseline as it has much lower number of parameters and hence much smaller gradients during back-propagation. Initialization and compilation overhead are not included in the results.

Results. Table 3 shows inference time (in milliseconds) and the runtime memory footprint (in megabytes) for the model using Weinberger's hashing trick and the models using our approach. `MEmCom` outperforms Weinberger's hashing trick for all computes on both smartphones. This is because our approach uses an efficient lookup operator described in section 3 and Weinberger's hashing method relies on the one-hot encoded representation. As CoreML and TF-Lite implement the lookup operator in the embedding layer using `mmap`, the memory footprint for `MEmCom` is very small compared to the Weinberger's hashing method. TF-Lite's `mmap` is tuned for lower memory footprint than for faster inference time. So we see a considerable difference in memory footprint, inference time betweeen CoreML and TF-Lite. The above results show that `MEmCom` has a very low-memory footprint and reasonably small inference time and hence is well suited to be deployed on off-the-shelf phones.

6 CONCLUSIONS

In this paper, we propose a novel method for compressing the embedding layer of a neural network without significant loss in the overall accuracy of the model. Unlike the hashing trick and similar methods, our method allows the network to learn a unique embedding vector for each categorical entity, giving the model an edge in the compression of large-scale search and recommendation models. We compared our approach with state-of-the-art model compression techniques on different problem classes and on multiple datasets. Our experiments validate that our approach significantly outperforms other techniques in terms of compression vs. accuracy tradeoffs. Furthermore, we benchmark the runtime performance of our approach with a state-of-the-art feature hashing technique, on an Apple iPhone 12 Pro and a Google Pixel 2 using popular libraries (CoreML and TensorFlow Lite). The results shows that our approach is well-suited for on-device inference; thereby paving the way for more privacy-friendly recommendation ecosystems.

REFERENCES

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

Apple. *Differential Privacy*, 2020. URL https://www.apple.com/privacy/docs/Differential_Privacy_Overview.pdf.

Bennett, J., Lanning, S., and Netflix, N. The netflix prize. 01 2009.

Bertin-Mahieux, T., Ellis, D. P., Whitman, B., and Lamere, P. The million song dataset. In *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*, 2011.

Bojanowski, P., Grave, E., Joulin, A., and Mikolov, T. Enriching word vectors with subword information. 2017.

Burges, C., Shaked, T., Renshaw, E., Lazier, A., Deeds, M., Hamilton, N., and Hullender, G. Learning to rank using gradient descent. In *Proceedings of the 22nd International Conference on Machine Learning*, ICML '05, pp. 89–96, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595931805. doi: 10.1145/1102351.1102363. URL https://doi.org/10.1145/1102351.1102363.

Chen, W., Wilson, J. T., Tyree, S., Weinberger, K. Q., and Chen, Y. Compressing neural networks with the hashing trick. *CoRR*, abs/1504.04788, 2015. URL http://arxiv.org/abs/1504.04788.

Denil, M., Shakibi, B., Dinh, L., Ranzato, M., and de Freitas, N. Predicting parameters in deep learning. *CoRR*, abs/1306.0543, 2013. URL http://arxiv.org/abs/1306.0543.

Denton, E., Zaremba, W., Bruna, J., LeCun, Y., and Fergus, R. Exploiting linear structure within convolutional networks for efficient evaluation. *CoRR*, abs/1404.0736, 2014. URL http://arxiv.org/abs/1404.0736.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. 2019.

Ginart, A., Naumov, M., Mudigere, D., Yang, J., and Zou, J. Mixed dimension embeddings with application to memory-efficient recommendation systems. *CoRR*, abs/1909.11810, 2019. URL http://arxiv.org/abs/1909.11810.

Gong, Y., Liu, L., Yang, M., and Bourdev, L. Compressing deep convolutional networks using vector quantization. 12 2014.

Grbovic, M., Djuric, N., Radosavljevic, V., and Bhamidipati, N. Search retargeting using directed query embeddings. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15 Companion, pp. 37–38, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334730. doi: 10.1145/2740908.2742774. URL https://doi.org/10.1145/2740908.2742774.

Gupta, U., Wu, C.-J., Wang, X., Naumov, M., Reagen, B., Brooks, D., Cottel, B., Hazelwood, K., Hempstead, M., Jia, B., Lee, H.-H. S., Malevich, A., Mudigere, D., Smelyanskiy, M., Xiong, L., and Zhang, X. The architectural implications of facebook's dnn-based personalized recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 488–501, 2020. doi: 10.1109/HPCA47549.2020.00047.

Han, S., Pool, J., Tran, J., and Dally, W. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 1135–1143, 2015.

Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *International Conference on Learning Representations (ICLR)*, 2016.

Harper, F. M. and Konstan, J. A. The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, 5(4), December 2015. ISSN 2160-6455. doi: 10.1145/2827872. URL https://doi.org/10.1145/2827872.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi: 10.1038/s41586-020-2649-2. URL https://doi.org/10.1038/s41586-020-2649-2.

He, R., Kang, W.-C., and McAuley, J. Translation-based recommendation. *Proceedings of the Eleventh ACM*

Conference on Recommender Systems, Aug 2017. doi: 10.1145/3109859.3109882. URL http://dx.doi.org/10.1145/3109859.3109882.

Hinton, G., Vinyals, O., and Dean, J. Distilling the knowledge in a neural network. In *NIPS Deep Learning and Representation Learning Workshop*, 2015. URL http://arxiv.org/abs/1503.02531.

Iverson, K. E. Notation as a tool of thought. *Commun. ACM*, 23(8):444–465, August 1980. ISSN 0001-0782. doi: 10.1145/358896.358899. URL https://doi.org/10.1145/358896.358899.

Jaderberg, M., Vedaldi, A., and Zisserman, A. Speeding up convolutional neural networks with low rank expansions. *CoRR*, abs/1405.3866, 2014. URL http://arxiv.org/abs/1405.3866.

Krishnamoorthi, R. Quantizing deep convolutional networks for efficient inference: A whitepaper. *CoRR*, abs/1806.08342, 2018. URL http://arxiv.org/abs/1806.08342.

Lan, Z., Chen, M., Goodman, S., Gimpel, K., Sharma, P., and Soricut, R. ALBERT: A lite BERT for self-supervised learning of language representations. *CoRR*, abs/1909.11942, 2019. URL http://arxiv.org/abs/1909.11942.

Le, H., Pham, Q., Sahoo, D., and Hoi, S. C. H. Urlnet: Learning a URL representation with deep learning for malicious URL detection. *CoRR*, abs/1802.03162, 2018. URL http://arxiv.org/abs/1802.03162.

Lecun, Y., Denker, J., and Solla, S. Optimal brain damage. volume 2, pp. 598–605, 01 1989.

Liu, B., Wang, M., Foroosh, H., Tappen, M., and Pensky, M. Sparse convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

Liu, Z., Sun, M., Zhou, T., Huang, G., and Darrell, T. Rethinking the value of network pruning. *CoRR*, abs/1810.05270, 2018. URL http://arxiv.org/abs/1810.05270.

McMahan, B., Moore, E., Ramage, D., Hampson, S., and y Arcas, B. A. Communication-efficient learning of deep networks from decentralized data. In Singh, A. and Zhu, X. J. (eds.), *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, AISTATS 2017, 20-22 April 2017, Fort Lauderdale, FL, USA*, volume 54 of *Proceedings of Machine Learning Research*, pp. 1273–1282. PMLR, 2017. URL http://proceedings.mlr.press/v54/mcmahan17a.html.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. Efficient estimation of word representations in vector space. 2013.

Mironov, I. Renyi differential privacy. *CoRR*, abs/1702.07476, 2017. URL http://arxiv.org/abs/1702.07476.

Murray, A. F. and Edwards, P. J. Enhanced mlp performance and fault tolerance resulting from synaptic weight noise during training. *IEEE Transactions on Neural Networks*, 5:792–802, 1994.

Nazari, Z., Charbuillet, C., Pages, J., Laurent, M., Charrier, D., Vecchione, B., and Carterette, B. Recommending podcasts for cold-start users based on music listening and taste. *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, Jul 2020. doi: 10.1145/3397271.3401101. URL http://dx.doi.org/10.1145/3397271.3401101.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019.

Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. Deep contextualized word representations. 2018.

Shi, H. M., Mudigere, D., Naumov, M., and Yang, J. Compositional embeddings using complementary partitions for memory-efficient recommendation systems. *CoRR*, abs/1909.02107, 2019. URL http://arxiv.org/abs/1909.02107.

Valizadegan, H., Jin, R., Zhang, R., and Mao, J. Learning to rank by optimizing ndcg measure. In *Proceedings of the 22nd International Conference on Neural Information Processing Systems*, NIPS'09, pp. 1883–1891, Red Hook, NY, USA, 2009. Curran Associates Inc. ISBN 9781615679119.

Vanhoucke, V., Senior, A., and Mao, M. Z. Improving the speed of neural networks on cpus. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*, 2011.

Vasile, F., Smirnova, E., and Conneau, A. Meta-prod2vec. *Proceedings of the 10th ACM Conference on Recommender Systems*, Sep 2016. doi: 10.1145/2959100.

2959160. URL http://dx.doi.org/10.1145/2959100.2959160.

Weinberger, K., Dasgupta, A., Langford, J., Smola, A., and Attenberg, J. *Feature Hashing for Large Scale Multitask Learning*, pp. 1113–1120. Association for Computing Machinery, New York, NY, USA, 2009. ISBN 9781605585161. URL https://doi.org/10.1145/1553374.1553516.

Wen, W., Wu, C., Wang, Y., Chen, Y., and Li, H. Learning structured sparsity in deep neural networks. *CoRR*, abs/1608.03665, 2016. URL http://arxiv.org/abs/1608.03665.

Xie, X., Lian, J., Liu, Z., Wang, X., Wu, F., Wang, H., and Chen, Z. Personalized recommendation systems: Five hot research topics you must know. https://www.microsoft.com/en-us/research/lab/microsoft-research-asia/articles/personalized-recommendation-systems/, Nov 2018.

Yin, C., Acun, B., Liu, X., and Wu, C. Tt-rec: Tensor train compression for deep learning recommendation models. *CoRR*, abs/2101.11714, 2021. URL https://arxiv.org/abs/2101.11714.

Zhang, C., Liu, Y., Xie, Y., Ktena, S. I., Tejani, A., Gupta, A., Myana, P. K., Dilipkumar, D., Paul, S., Ihara, I., Upadhyaya, P., Huszar, F., and Shi, W. Model size reduction using frequency based double hashing for recommender systems. In *Fourteenth ACM Conference on Recommender Systems*, RecSys '20, pp. 521–526, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375832. doi: 10.1145/3383313.3412227. URL https://doi.org/10.1145/3383313.3412227.

Zhu, C., Han, S., Mao, H., and Dally, W. J. Trained ternary quantization. *CoRR*, abs/1612.01064, 2016. URL http://arxiv.org/abs/1612.01064.

A Additional Experimental Evaluation

The section 5 focused on evaluating the effectiveness of MEmCom for a typical on-device inference scenario. In this section, we expand on that to cover more advanced scenarios.

A.1 Fixed model size

Experimental Setup. In the section 5, we assumed that the hyperparameters of the baseline uncompressed model was tuned to achieve best generalization and the goal was to compress the model as much as possible with minimal loss of accuracy. To that end, we evaluated MEmCom under different number of embeddings (i.e. hash sizes), while assuming that the embedding size was the model hyperparameter. This is suitable for most cases where the data scientist chooses a reasonably large embedding size; which in our case was 256. In this experiment, we fix the model size and vary both the embedding size as well as the number of embeddings. This experiment evaluates a scenario where a size budget is imposed by the use-case and the goal is to find the best set of hyperparameters that satisfy that budget. Also, this experiment helps us understand the tradeoff between chosing a large embedding size (and a small number of embeddings) and a large number of embeddings (and a small embedding size) for MEmCom. For the public datasets, we fixed the size of each model to half the size of the corresponding baseline model. For the Arcade and Games datasets, we fixed the model size to be 20MB. Other than that, the experimental setup is same as the one described in the section 5.1.

If we increase the number of embeddings, we have to decrease the embedding size accordingly to ensure that model size remains the same. As the model size also depends on the output vocabulary size, we performed a simple binary search to find the embedding size for corresponding number of embeddings.

Results. The plot 6 shows the tradeoff between choosing a larger embedding size vs larger number of embeddings. It shows that for most use cases, including Millionsongs, Movielens, Netflix, Games and Arcade, the optimal number of embeddings for MEmCom is roughly 10x lower than its input vocabulary. Interestingly, this did not hold for the Google Local Reviews use case, where the distribution of reviews is more even across all entities due to geographical constraints.

A.2 Lower precision

Experimental Setup. As mentioned in the section 2, we can reduce the size of a model compressed via MEmCom by either reducing the floating point precision of weights and by sparsifying the weights (Han et al., 2015; 2016; Murray & Edwards, 1994; Vanhoucke et al., 2011; Krishnamoorthi, 2018). We leave the latter as a future work and focus on the effectiveness of quantization for model compression. We use the model described in the section A.1 and evaluate the loss in accuracy compared to a model that is compressed using MEmCom and whose weights are stored in single-point precision (i.e. 32 bits). By reducing the floating point precision by half, we reduce the model size by half. Hence, we use the floating point precision as x-axis to compare the effect of quantization across multiple datasets. We evaluate the results using CoreML[3] and with the quantization mode set to `linear`.

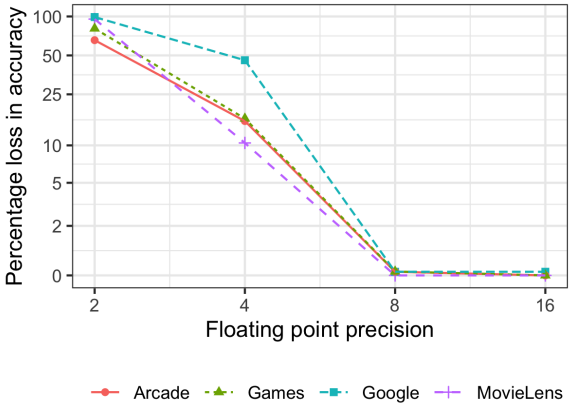

Figure 4: Accuracy vs floating point precision tradeoff.

Results. As shown in the figure 4, all the datasets (except Google Local Review) have no loss in accuracy when the model is converted to half-point precision. The model trained on MovieLens can be compressed even further using 8-bit precision without any loss in accuracy. On all other datasets, the loss of accuracy is approximately 0.13% when using 8-bit precision. This implies that a typical recommender model can be compressed further by 4x using 8-bit precision with negligible loss in accuracy. However, the accuracy drops significantly if we quantize the model further for all the datasets.

A.3 Robustness to noise for private federated learning

Experimental Setup. Models deployed on-device have greater access to user data. This data is typically not sent to the server to protect user's privacy. To train such a model, one uses private federated learning, where the models are trained in a distributed manner across multiple devices. To guarantee user privacy, "noise" is introduced during training which makes it difficult to infer whether any particular data

[3] https://developer.apple.com/machine-learning/core-ml/

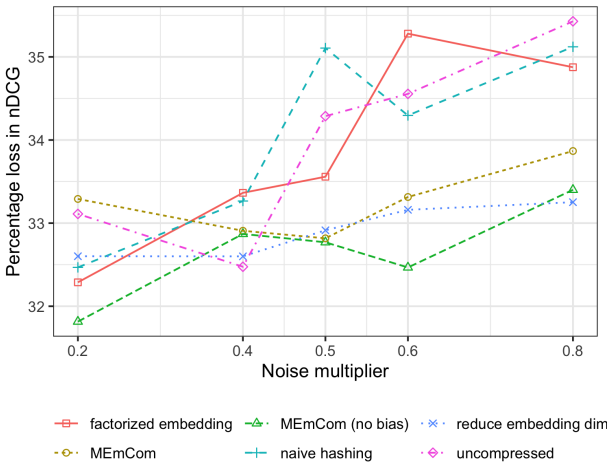

Figure 5: Evaluating privacy vs accuracy tradeoff: Arcade

points was used during training. However, this may degrade model accuracy compared to one trained without noise. In this experiment, we focus on the tradeoff robustness of model compression techniques vs. noise. We use an uncompressed model without noise as a baseline and evaluate percentage loss in nDCG for a model trained with a given noise multiplier. We use the same dataset as described in the section 5.2. Other than `reduce embedding dim`, we set the hyperparameters such that the compressed models were 51 MB in size. The model compressed using `reduce embedding dim` was 102 MB in size. To simulate differential private federated learning, we trained the models with the Rényi Differential Privacy (RDP) framework (Mironov, 2017; Apple, 2020) using the TensorFlow Privacy library. We use used global DP setup, constant l2_norm_clip. and set RDP's δ parameter to $\frac{1}{number\ of\ training\ points}$. Since the TensorFlow Privacy library is only supported for TensorFlow 2.0 or greater, we used TensorFlow 2.4.0.

Results. The plot 5 compares the relative accuracy (in our case nDCG) of different approaches to that of an uncompressed model trained without noise for different noise multipliers. It shows that our approach has lower loss in nDCG for a given noise multiplier and was more robust to noise that an uncompressed model and naive hashing.

A.4 Practical sanity check that MEmCom produces unique embeddings

Experimental Setup. On one model that was trained on the Arcade dataset using MEmCom, with an input embedding compression ratio of 40x, we examined the uniqueness of the embeddings that were produced. This experiment allowed us to test our claim that MEmCom is able to produce a unique embedding for each category.

Results. We found that a vanishingly small number of categories that shared an $\mathbf{x}_{rem}$ embedding, as defined in Algorithm 2, ended up with equal $\mathbf{x}_{mult}$ multipliers. More precisely, a pair of multipliers sharing a common $\mathbf{x}_{rem}$ embedding differed by greater than 0.00001 in more than 99.98% of cases. This result validates the claim that MEmCom is able to produce a unique embedding for each category.

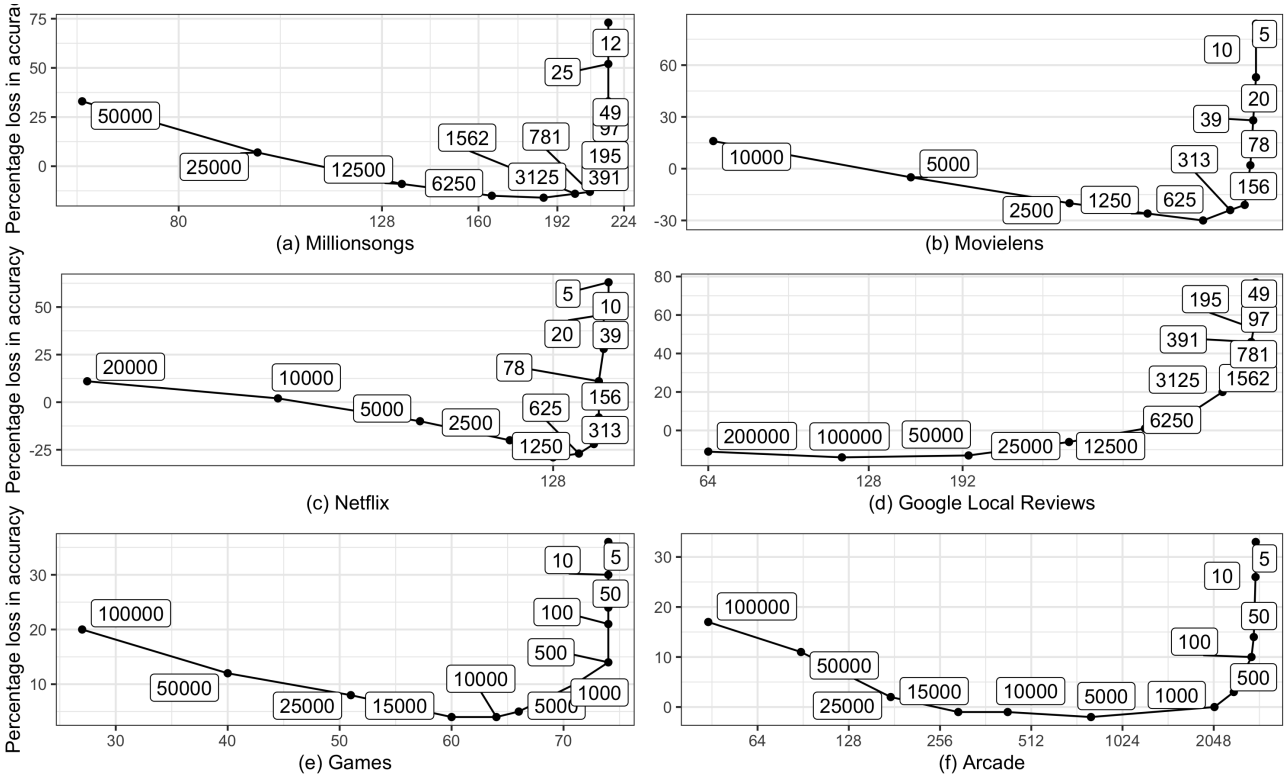

Figure 6: Tuning the embedding size for given model size (= 20 MB). The x-axis denotes the embedding sizes and each datapoint is annotated with the corresponding number of embeddings.