December 2019

Dot Product Matrix Compression for Machine Learning

Anonymous

Dot Product Matrix Compression for Machine Learning

ABSTRACT

When pairwise dot products are computed between input embedding vectors and the dot product is used for further computation, the number of dot products grows quadratically with the number of embedding vectors. This can cause an efficiency bottleneck and affect performance of machine learning models. This disclosure describes techniques to obtain a compressed dot product matrix from input sparse embeddings. The compressed embeddings are used to obtain a compressed dot product. The compressed embeddings are generated using a weights matrix that is initialized randomly and learnt alongside other parts of the model. To improve performance, attention weights derived from the input embeddings can be used as the weights matrix. Still further, a high level representation of the input embeddings can be obtained and combined with a low-level representation. The described compression techniques can improve model accuracy, as measured by normalized entropy and can improve model execution efficiency. The reduction in size of the dot product matrix, enabled by the described techniques, reduces computational complexity.

KEYWORDS

- Dot product
- Bottleneck layer
- Low rank approximation
- Attention weights
- Machine learning
- SparseNN
- Resnet

BACKGROUND

A neural network accelerator that exploits sparsity, such as SparseNN [1], by default calculates all pairwise dot products between each pair of embedding vectors in the input and passes the dot products to the fully connected (FC) layers in over-arch. As a result, the number of dot products grow quadratically in terms of the number of embedding vectors. With increasingly large over-archs, the connection of dot products with FC layer can become an efficiency bottleneck and can constrain model capacity.

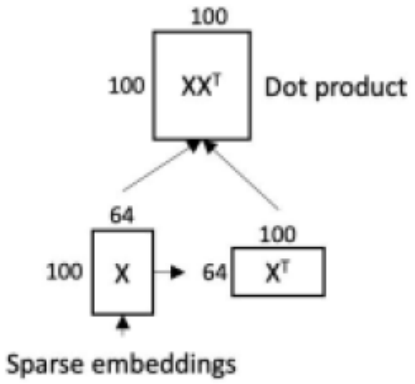

Fig. 1: Dot product computation from input embeddings

Fig. 1 illustrates dot product computation. As can be seen in Fig. 1, X is an n*d matrix of n d-dimensional embedding vectors. In the example of Fig. 1, n= 100 and d=64. The resultant dot product - XX^T - has dimensions n*n.

A few optimization techniques can be commonly applied to mitigate this issue. For example, the embedding vectors can be divided into several disjoint groups, based on the underlying data. After the division, the calculation of dot products within a group is skipped. Another approach is to estimate the importance of each dot product and feed only the most important ones into the FC layers. Such optimizations can increase the queries-per-second (QPS) that can be processed due to the reduced number of dot products that are sent to the over-archs.

However, these optimizations require non-trivial human inputs or extra processing to determine which dot products to remove. This can be a costly step when adding new sparse features. Also, since there are dot products that are removed, it is possible that useful information is lost.

DESCRIPTION

Compression of Dot Product

This disclosure describes techniques to compress a dot product matrix of size n*n to a matrix of size n*k, where n is the number of embedding vectors and k <= n is a parameter that controls the compression ratio. The compression is performed prior to passing the matrix to the over-arch. The technique is illustrated in Fig. 2.

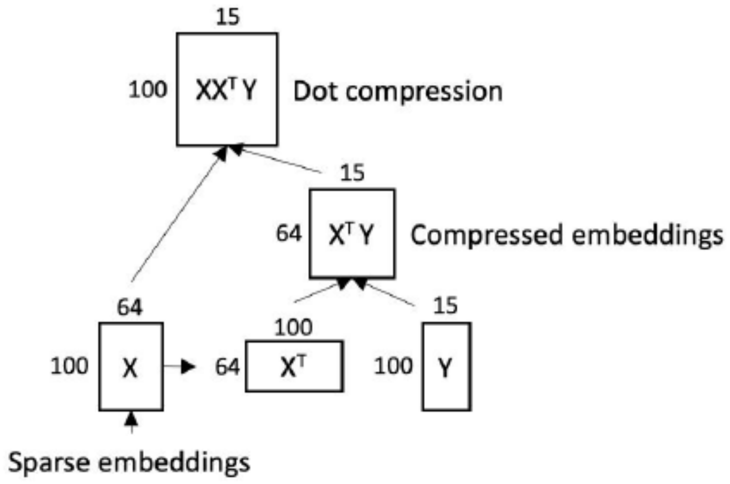

Fig. 2: Compression of dot product

In the example of Fig. 2, X is an n*d matrix of n d-dimensional embedding vectors (n=100;d=64). Per techniques described herein, instead of calculating flatten(XX^T), flatten(XX^TY) is calculated. Y is an n*k matrix (in the example of Fig. 2, k=15). Y is a learnable parameter and is initialized randomly. Y can be learnt, e.g., using stochastic gradient descent, alongside other parts of the model. As seen in Fig. 2, an output vector of size $n*k < n*n$ for $k < n$ is obtained. In the example of Fig. 2, the size of the output vector is 100*15.

Given n sparse features, each sparse feature is represented by an embedding of size d. The sparse embeddings are represented by an n*d matrix X. The dot products between sparse embeddings are represented by $X(X^T)$, which is an n*n matrix.

Per techniques described herein, k linearly compressed embeddings are leant, each of which is a weighted sum of raw embeddings. The compressed embeddings can be represented by a d*k matrix $(X^T)Y$, where Y is an n*k weights matrix. Dot products are calculated between these compressed embeddings and raw embeddings (X) to obtain an n*k matrix $X(X^T)Y$. In this technique, since the number of dot products is reduced from n^2 to n*k, a lot of computations can be saved in the first FC layer overarch.

The projection through Y significantly reduces the number of neurons passing to the over-arch while still preserving the information of original dot product matrix by exploiting the low rank property of the dot product matrix. In addition, by computing X^TY first, the complexity of computing compressed dot products is O(ndk), which is faster than the complexity $O(n^2d)$ of computing all dot products when $k < n$.

The compression technique exploits the fact that the dot product matrix XX^T has rank d when $d <= n$, where d is the dimensionality of embedding vectors. Thus, XX^T is a low rank matrix that has O(nd) degree of freedom rather than O(n^2). In other words, XX^T is compressible. This is true for many current model types that have sparse input data. This allows compression of the dot product without loss of information.

Dimensionality reduction via linear projection: The compression technique described with reference to Fig. 2 is essentially an extension of random linear projection. The matrix XX^T

consists of n row vectors and each vector has n dimensions. Since XX^T has rank d, these n row vectors are contained in a d-dimensional subspace. Random linear projection compresses the vectors to O(d) dimensions such that if the original vectors are in a low-dimension subspace, the pairwise distances between them are preserved after compression. In the present context, it works by right multiplying the matrix XX^T by a random Gaussian matrix (up to a scaling factor) Y and XX^TY approximates XX^T. When implementing compression, Y may be learnt, e.g., after random initialization. This makes it possible to identify important dimensions during the compression.

Feature compression: The compression technique can be also considered as a way to compress the feature vectors. The compressed dot product matrix can be written as $X(X^TY)$. The d*k matrix X^TY can be seen as k d-dimensional “compressed features” (or representative features), and the result can be considered as the pairwise dot products between the original feature vectors and the compressed ones. When Y is a trained alongside with the rest of neural network, the learning algorithm can find a good and compact representation of the original features by extracting the important ones and reducing redundancy among the original features.

Compression can produce consistent QPS gain for models with large over-archs than when using dot product (Fig. 1) or when grouping of embedding vectors is used. The reduction in the number of dot products make computation of the FC layer on the dot products less expensive. It can also produce consistent improvements in normalized entropy (NE) gain. When adding more sparse features, the NE gain brought by compression is higher. This can be, at least in part, due to the automatically learnt compression better preserving the dot product information than manual optimizations such as grouping. In some experiments, dot compression can also outperform baselines with full dot products (no grouping). This can be due to learning X^TY

which can reduce the redundancy among features or remove noise within input signals. The technique is applicable to other model types with minimum tuning efforts, e.g., models with a medium to large number of sparse features.

Dot compression, as described above, first linearly compresses (weighted sums) the sparse embeddings and then computes the dot products between these compressed embeddings and raw embeddings. However, linear compression can only learn low-level representations and is just a weighted sum of raw embeddings.

Below, two architectures are described that learn non-linear and high level representations from features.

Compression with attention weights

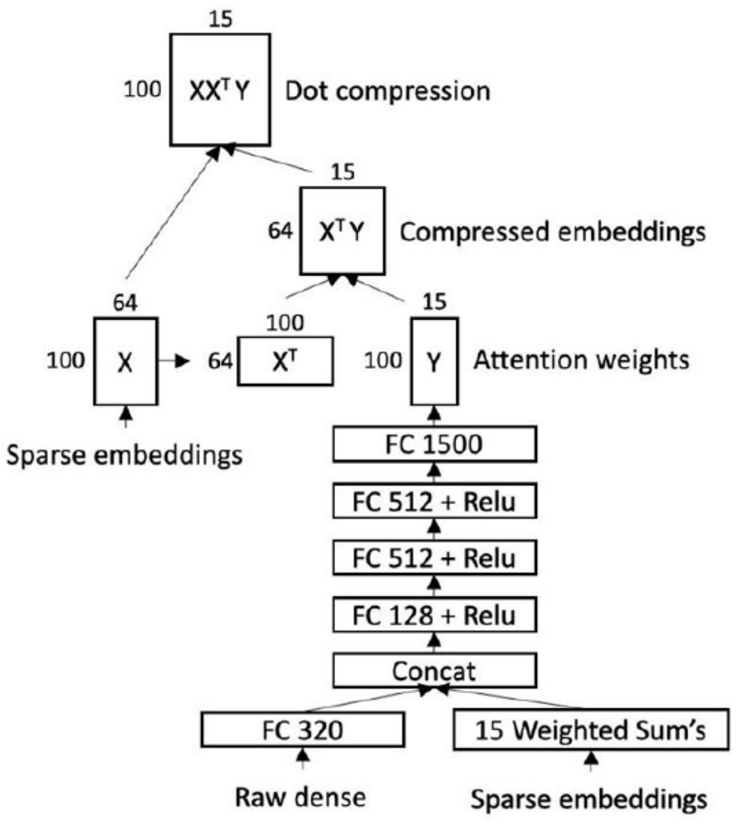

Fig. 3: Example of compression with attention weights

A drawback of the compression technique described with reference to Fig. 2 above is that it uses the same weights matrix Y for each example to compress raw embeddings. While it is possible to learn an element-wise attention weights matrix on top of $X(X^T)Y$ which allows the model to focus on different parts of the dot compression output for different examples, Y is still the same for each example, which means the way the raw embeddings are compressed remains the same.

Fig. 3 illustrates an architecture to apply attention technique directly to the weights matrix Y. The architecture allows the model use different weights to linearly compress raw embeddings for different examples. Given the raw dense features D and raw sparse embeddings X, Y is defined as

Y = MLP(Concat(LC(D), LC(X))), where

LC(*) means doing linear compression (weighted sum) for *,

Concat(*) is a concat operator for inputs, and

MLP(*) is to apply Multilayer Perceptron for input *.

In the formula Y = MLP(Concat(LC(D), LC(X))), the linear compression LC(D) = FC(D) for dense features D, where FC(*) means a fully connected layer without non-linear function; and LC(X) are a set of weighted sums of the sparse embeddings in X. The use of LC(*) before Concat(*) reduces the input size for MLP(*) and improves model efficiency. The output of MLP(*) is reshaped before use. As a result, if Y is a 100*15 weights matrix, then the output size of MLP(*) is set at 1500. While Fig. 3 shows a specific configuration of fully connected (FC)/Rectified Linear Unit (ReLU) layers in the computation of attention weights, the configuration can be adjusted, e.g., to use fewer or more layers for different input embeddings.

Using the architecture described with reference to Fig. 3, different weights matrix Y can be generated for different examples from their dense features and raw sparse embeddings. Essentially, the example drives the compression of sparse embeddings based on the information from its features.

This architecture enables learning the weights for compressing sparse embeddings with attention mechanism to boost relevant, important features and reduce the redundancy of original features. The attention mechanism takes raw dense features and summarization of sparse features with compressed embeddings as input to learn the corresponding weights.

Resnet-style compression

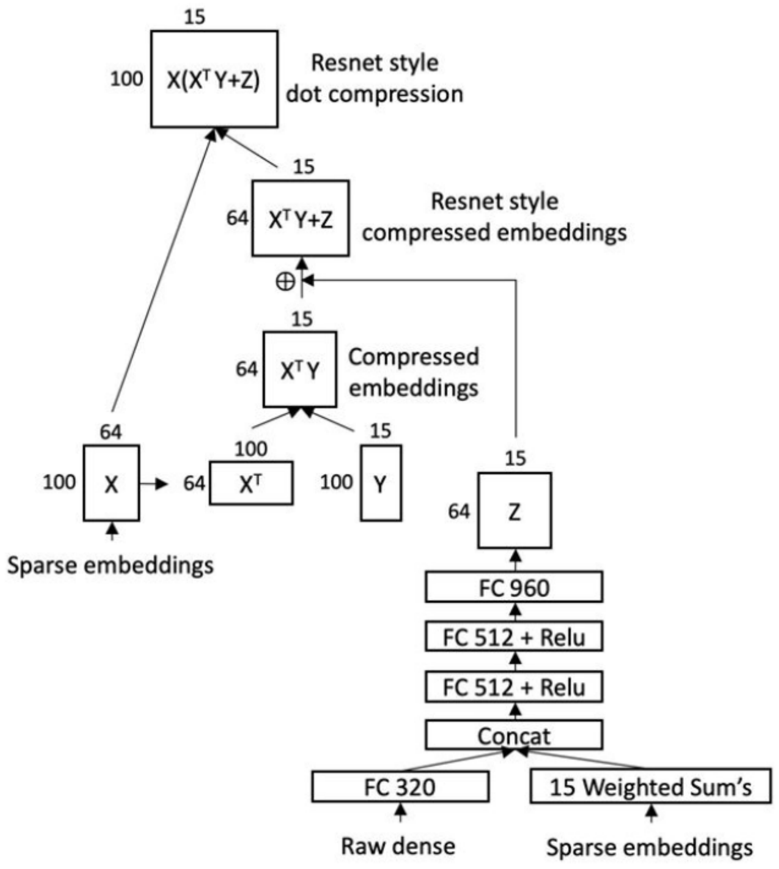

Fig. 4: Resnet-style compression

In this architecture, the linearly compressed embeddings $(X^T)Y$ are first computed, where Y is a normal weights matrix. These are then used to obtain $(X^T)Y + Z$ as the new/improved compressed embeddings, where Z = MLP(Concat(LC(D), LC(X))). The linear compression for dense features LC(D) and sparse embeddings LC(X) reduces the computational cost of the first FC layer in MLP(*). The output of last layer of MLP(*) is reshaped before using it as Z and Z has the same shape as $(X^T)Y$ such that these can be element-wisely added together. For example, if $(X^T)Y$ is a 64-by-15 matrix, then the output size of MLP(*) is 960. Finally, the dot products between these new compressed embeddings and raw sparse embedding are computed as $X[(X^T)Y + Z]$ which is the compression output.

This architecture uses a skip connection, similar to Resnet. $(X^T)Y$ is a set of linearly compressed embeddings (LCE), and the "15 Weighted Sums" above is another set of LCE. Therefore, $(X^T)Y + Z$ = LCE + MLP(Concat(LC(D), LCE')). This is similar to a Resnet skip connection from LCE to MLP(LCE), although two LCE are actually different. While Fig. 4 shows a specific configuration of fully connected (FC)/Rectified Linear Unit (ReLU) layers in the computation of attention weights, the configuration can be adjusted, e.g., to use fewer or more layers for different input embeddings.

In this architecture, LCE is a low-level representations of sparse embeddings, and MLP(Concat(LC(D), LCE')) is a high-level representation, due to the Multilayer perceptron. These two hidden layers capture meaningful information from different levels and are combined by element-wise addition. This kind of skip connection can facilitate the learning in a deep neural network due to its "identity mapping + residual" structure.

Resnet style compression with attention weights

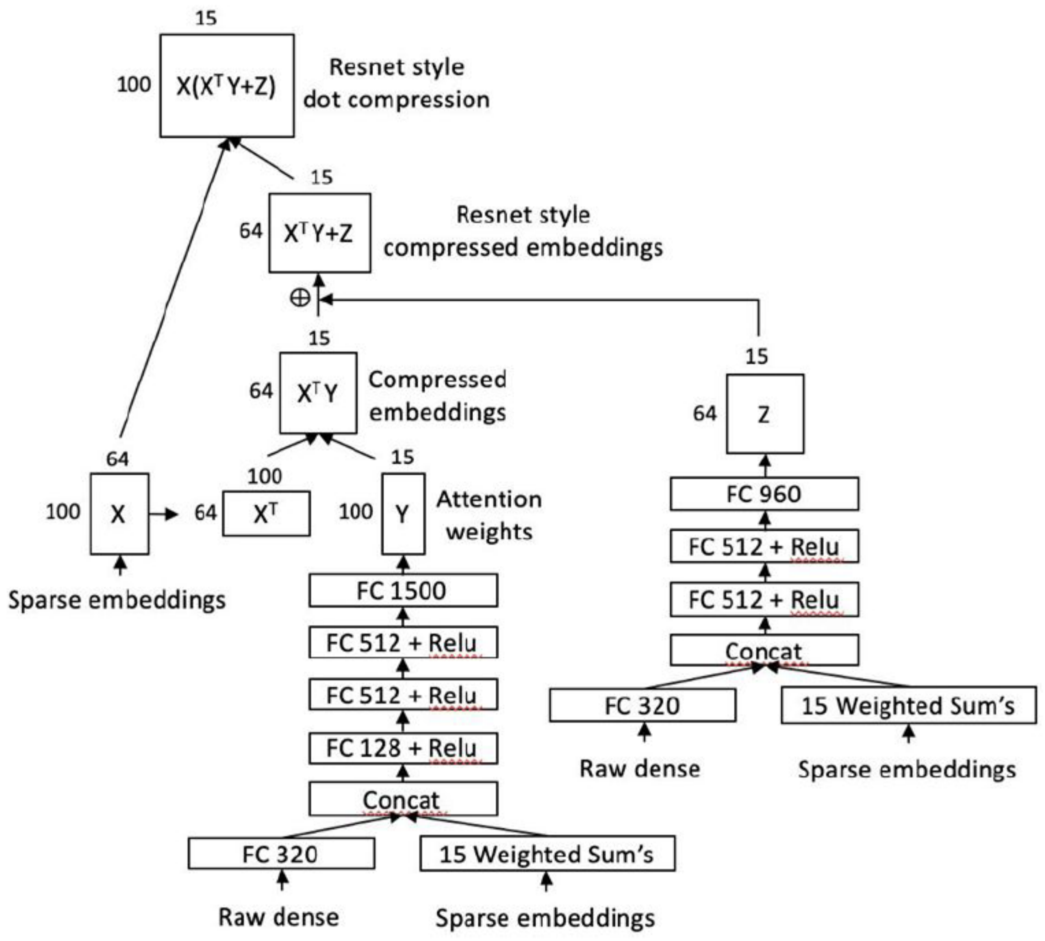

Fig. 5: Resnet-style compression with attention weights

The two architectures described above can be combined, as illustrated in Fig. 5. In this setup, MLP(*) is used to obtain the attention weights matrix Y. Another MLP(*) is used to get the "residual" Z in skip connection. The final output of this architecture can be written as $X[(X^T)Y + Z]$, where Y = MLP(Concat(LC(D), LC(X))) and Z = MLP'(Concat(LC'(D), LC'(X))).

The combination of architectures used in Fig. 5 makes $(X^T)Y$ deeper and more like a residual, compared to Z. To make it more like a regular skip connection, instead of naively combining these together, it may be possible to add a skip connection layer on top of (X^T)Y, or on top of Z.

The architecture described with reference to Fig. 5 enables learning both low-level and high-level representations from raw embeddings and combines them by a skip connection. Training speed is improved by capturing non-linearity through MLP on top of raw dense features and by summarization of sparse features with compressed embeddings.

The compression techniques described herein with reference to Figs. 3-5 can be applied in various machine learning applications. In actual use, parameters such as the maximum cardinality of dense features, the value of k, and the number/size of embeddings for each sparse feature can be selected as appropriate.

Advantages

The described techniques for dot product compression can improve both model quality and machine learning efficiency.

- **Normalized entropy (NE) Gain:** By using a learnable approximation of the full dot product matrix, the described technique is able to achieve better NE compared to manual selection and grouping of a subset of dot products.
- **Efficiency:** With a proper compression size, the technique can significantly increase efficiency due to the reduced number of neurons sending to the overarch and allows adding more sparse features.
- **Development Efficiency:** The technique can automatically extract useful feature interactions, thus enabling reduction in manual optimization.

CONCLUSION

This disclosure describes techniques to obtain a compressed dot product from input sparse embeddings. Specifically, compressed embeddings are used to obtain a compressed dot product. The compressed embeddings are generated using a weights matrix that is initialized randomly and learnt alongside other parts of the model. To improve performance, attention weights derived from the input embeddings can be used as the weights matrix. Still further, a high level representation of the input embeddings can be obtained and combined with a low-level representation. The described compression techniques can improve model accuracy, as measured by normalized entropy and can improve model execution efficiency. The reduction in size of the dot product matrix, enabled by the described techniques, reduces computational complexity.

REFERENCES

1. Zhu, Jingyang, Jingbo Jiang, Xizi Chen, and Chi-Ying Tsui. "SparseNN: An energy-efficient neural network accelerator exploiting input and output sparsity." In 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 241-244. IEEE, 2018.