Chapter 4

Sparse Tensors Networks

4.1 Introduction

Compressing a neural network to speedup inference and minimize memory footprint has been studied widely. One of the popular techniques for model compression is pruning the weights in convnets, is also known as sparse convolutional networks [9, 11]. Such parameter-space sparsity used for model compression compresses networks that operate on dense tensors and all intermediate activations of these networks are also dense tensors.

However, there is another type of sparsity that is widely used for high-dimensional data: spatial sparsity. Natural language and images tend to occupy the spatial dimensions of the modality densely, but in 3-dimensional scans or higher-dimensional statistical data, the data occupy a small portion of the space and are sparse. For example, in Fig. 4.1, we visualize an example 3D scan of a hotel room from the ScanNet dataset [4] with various quantization step sizes. As we increase the spatial resolution or decrease the quantization step size, the sparsity of the volume decreases rapidly.



Figure 4.1: The sparsity ratios of a 3D scan with varying quantization size. As we increase the spatial resolution of the space, the sparsity of the data decreases rapidly.

Such spatially sparse data are commonly used to represent 3D point clouds or meshes as 3D reconstructions from images or scans from Time-of-Flight scanners generate an unstructured set of points in the 3D space. Likewise, the majority of machine learning problems and statistical data are defined in a high-dimensional space and occupy an extremely small portion of the space. Thus, as we increase the spatial resolution or the dimension of the space, the sparsity of the data decreases rapidly and representing such sparsity is the key component that allows efficient learning.

One of the first families of neural networks that process such spatially sparse data is spatiallysparse convolutional neural networks [5, 8, 2, 3], some of the spatially-sparse networks take spatially sparse tensors and some of the intermediate activations are sparse tensors as well.

We define such neural networks specialized for spatially sparse tensors as **sparse tensor net-works**; sparse tensor networks take a sparse tensor as input, process, and generate sparse tensors as activations and output. To construct a sparse tensor network, we build all standard neural network layers such as MLPs, non-linearities, convolution, normalizations, pooling operations in the same way we define them on a dense tensor and implemented in the Minkowski Engine [2].

4.2 Sparse Tensor Networks

In this section, we delineate the general operation pipeline of sparse tensor networks and some common layers and terminologies used for sparse tensor networks. Also, we formally define convolution layers, pooling layers, normalization layers, and non-linearities used in sparse tensor networks. These operations are derived from the same counterparts in a conventional neural network, and work similarly, but one of the major differences between sparse tensor networks and conventional networks is the sparsity management. In dense tensors, random access on an arbitrary element is easy to implement, but in sparse tensors, random access requires a relatively complex data structure such as a hash-table or KD-tree. Thus, searching neighboring points required for convolution and max-pooling becomes a non-trivial operation that requires managing data structures for sparsity patterns and tracking neighbors between them.

4.2.1 Terms

First, we cover some terms in sparse tensor networks and introduce basic operations that are critical for the generative sparse tensor network. Throughout the section, we will use lowercase letters for variable scalars, t; uppercase letters for constants, N; lowercase bold letters for vectors, \mathbf{v} ; uppercase bold letters for matrices, \mathbf{R} ; Euler scripts for tensors, \mathcal{T} ; and calligraphic symbols for sets, \mathcal{C} . We denote *i*-th row and *j*-th column element of a matrix \mathbf{A} as A[i, j] and the *i*-th row-slice as $\mathbf{A}[i, :]$.

Sparse Tensor

A tensor is a multi-dimensional array that represents high-order data. A *D*-th order tensor \mathcal{T} requires D indices to uniquely access its element and we denote such indices or a coordinate as $[x^1, ..., x^D]$ and the element as $\mathcal{T}[x^1, ..., x^D]$ similar to how we access components in a matrix. Likewise, a sparse tensor is a high-dimensional extension of a sparse matrix where the majority of the elements are 0

$$\mathcal{T}[x_i^1, x_i^2, \cdots, x_i^D] = \begin{cases} \mathbf{f}_i & \text{if } (x_i^1, x_i^2, \cdots, x_i^D) \in \mathcal{C} \\ 0 & \text{otherwise} \end{cases}$$
(4.1)

where C is the set of coordinates where we have non-zero values and \mathbf{f}_i is the non-zero value at *i*-th coordinate $\mathbf{x}_i = [x_i^1, x_i^2, \cdots, x_i^D]^T$.

We can represent the above sparse tensor compactly with two matrices: one for coordinates of non-zero values and another matrix for the non-zero values \mathbf{f} . This representation is simply rowwise concatenation of the coordinates $\mathbf{C} = [\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N]^T$ and values $\mathbf{F} = [\mathbf{f}_1, \mathbf{f}_2, ..., \mathbf{f}_N]^T$ and is also known as the COOrdinate list (COO) format [12].

$$\mathbf{C} = \begin{bmatrix} x_1^1 & x_1^2 & \cdots & x_1^D \\ \vdots & \vdots & \ddots & \vdots \\ x_N^1 & x_N^2 & \cdots & x_N^D \end{bmatrix}, \ \mathbf{F} = \begin{bmatrix} \mathbf{f}_1^T \\ \vdots \\ \mathbf{f}_N^T \end{bmatrix}$$
(4.2)

In sum, a sparse tensor is a multi-dimensional array whose majority of elements are 0. The COO representation of a sparse tensor uses a set of coordinates C or equivalently a coordinate matrix $C \in \mathbb{Z}^{N \times D}$ and associated features \mathcal{F} or a feature matrix $F \in \mathbb{R}^{N \times N_F}$ to represent the sparse tensor, where N is the number of non-zero elements within a sparse tensor, D is the dimension of the space, and N_F is the number of channels. We will use the COO representation to represent a sparse tensor throughout the chapter.

Tensor Stride

Receptive field size of a neuron is defined as the maximum distance along one axis between pixels in the input (image) that the neuron in a layer can see. For example, if we process an image with two convolution layers with kernel size 3 and stride 2, the receptive field size after the first convolution layer is 3; and the receptive field size after the second convolution layer is 7. This is due to the fact that the second convolution layer sees the feature map that subsamples the image with the factor of 2 or stride 2. Here, the stride refers to the distance between neurons. The feature map after the first convolution has the stride size 2 and that after the second convolution has the stride size 4. Similarly, if we use transposed convolutions (deconv, upconv), we reduce the stride.

We define a tensor stride to be the high-dimensional counterpart of these 2-dimensional strides

in the above example. When we use pooling or convolution layers with stride greater than 1, the tensor stride of the output feature map increases by the factor of the stride of the layer.

Kernel Map

A sparse tensor consists of a set of coordinates $\mathbf{C} \in \mathbb{Z}^{N \times D}$ and associated features $\mathbf{F} \in \mathbb{R}^{N \times N_F}$ where N is the number of non-zero elements within a sparse tensor, D is the dimension of the space, and N_F is the number of channels. To find how a sparse tensor is mapped to another sparse tensor using spatially local operations such as convolution or pooling, we need to find which coordinate in the input sparse tensor is mapped to which coordinate in the output sparse tensor.

We call this mapping from an input sparse tensor to an output sparse tensor a kernel map. For example, a 2D convolution with kernel size 3 has a 3×3 convolution kernel, which consists of 9 weight matrices. Some input coordinates are mapped to corresponding output coordinates with each kernel. We represent a map as a pair of lists of integers: the in map I and the out map O. An integer in an in map $i \in \mathbf{I}$ indicates the row index of the coordinate matrix or the feature matrix of an input sparse tensor. Similarly, an integer in the out map $o \in \mathbf{O}$ also indicates the row index of the coordinate matrix of an output sparse tensor. The integers in the lists are ordered in a way that k-th element i_k in the in map corresponds to the k-th element o_k of the out map. In sum, $(\mathbf{I} \to \mathbf{O})$ defines how the row indices of input feature F_I maps to the row indices of output feature F_O .

Since a single kernel map defines a map for one specific cell of a convolution kernel, a convolution requires multiple kernel maps. In the case of a 3×3 convolution in this example, we need 9 maps to define a complete kernel map.



Figure 4.2: An example kernel map for 3×3 convolution kernel. The 0-th input coordinate will be mapped to 0-th output coordinate through kernel I, I: $0 \rightarrow 0$. Similarly, B: $1 \rightarrow 0$, B: $0 \rightarrow 2$, D: $3 \rightarrow 1$ and H: $2 \rightarrow 3$.

4.2.2 Sparse Tensor Network Layers

A sparse tensor network consists of layers that take a sparse tensor as input and returns a sparse tensor as output. We formally define some of the most commonly used layers in this section.

Generalized Convolution

Conventional convolutions used in image processing, speech, and machine translation, are discrete convolution where the input and output are discrete dense tensors. Let $f_{\mathbf{u}}^{\mathrm{in}} \in \mathbb{R}^{N^{\mathrm{in}}}$ be an input feature with N^{in} channels in a *D*-dimensional discrete space located at $\mathbf{u} \in \mathbb{Z}^D$ (a *D*-dimensional coordinate), and convolution kernel weights be $\mathbf{W} \in \mathbb{R}^{K^D \times N^{\mathrm{out}} \times N^{\mathrm{in}}}$ where *K* is the kernel size. We break down the weights into K^D matrices and denote them as $W_{\mathbf{i}} \in N^{\mathrm{out}} \times N^{\mathrm{in}}$ for $i \in \mathcal{V}^D(K)$ and $\|\mathcal{V}^D(K)\| = K^D$. We can summarize the conventional dense convolution in D-dimension as

$$\mathbf{f}_{\mathbf{u}}^{\text{out}} = \sum_{\mathbf{i} \in \mathcal{V}^D(K)} W_{\mathbf{i}} \mathbf{f}_{\mathbf{u}+\mathbf{i}}^{\text{in}} \text{ for } \mathbf{u} \in \mathbb{Z}^D,$$
(4.3)

where $\mathcal{V}^D(K)$ is the list of offsets in a *D*-dimensional hypercube centered at the origin. e.g. $\mathcal{V}^1(3) = \{-1, 0, 1\}$.

A sparse tensor network uses sparse tensors as input and activations. We propose a generalized convolution that encompasses the conventional convolution as well as convolution on sparse tensors [6, 7]. Specifically, we generalize convolution to incorporate generic input and output coordinates and arbitrary kernel shapes. The generic input and output coordinates allow the generalized convolution to cover the dense convolution in Eq. 4.3 as well as the convolution on sparse tensors. In addition, it allows the network to handle asks that require generating coordinates dynamically such as 3D reconstruction and completion. Mathematically, we represent the generalized convolution as

$$\mathbf{f}_{\mathbf{u}}^{\text{out}} = \sum_{\mathbf{i} \in \mathcal{N}^{D}(\mathbf{u}) \cap \mathcal{C}^{\text{in}}} W_{\mathbf{i}} \mathbf{f}_{\mathbf{u}+\mathbf{i}}^{\text{in}} \text{ for } \mathbf{u} \in \mathcal{C}^{\text{out}}$$
(4.4)

where \mathcal{N}^D indicates the set of offsets that defines the shape of the convolution kernel. Specifically, $\mathcal{N}^D(\mathbf{u}) \cap \mathcal{C}^{\text{in}}$ is the intersection of the kernel offsets from the current point \mathbf{u} with the input coordinates \mathcal{C}^{in} . \mathcal{C}^{in} and \mathcal{C}^{out} are predefined input and output coordinates. Note that first, \mathcal{C}^{out} can be generated dynamically which is crucial for generative tasks. Second, the output coordinates can be defined arbitrarily independent of input coordinates. Third, the shape of the convolution kernel can be defined arbitrarily \mathcal{N}^D .



Convolution on a Sparse Tensor

Figure 4.3: Schematics of convolution on a dense tensor (top) and a sparse tensor (bottom).

Max Pooling and Global Pooling

Max pooling layer selects the maximum element within a region for each channel. For a sparse tensor input, we define it as

$$\mathbf{f}_{\mathbf{u},i}^{\text{out}} = \max_{\mathbf{k} \in \mathcal{N}^{D}(\mathbf{u}) \cap \mathcal{C}^{\text{in}}} \mathbf{f}_{\mathbf{u}+\mathbf{k},i}^{\text{in}}$$
(4.5)

where $\mathbf{f}_{\mathbf{u},i}$ indicates the *i*-th channel feature value at \mathbf{u} . The region to pool features from is defined as $\mathcal{N}^D(\mathbf{u}) \cap \mathcal{C}^{\text{in}}$. The global pooling is similar to Eq. 4.5 except that we pool features from all non-zero elements in the sparse tensor.

$$\mathbf{f}_{i}^{\text{out}} = \max_{\mathbf{k}\in\mathcal{C}^{\text{in}}} \mathbf{f}_{\mathbf{u}+\mathbf{k},i}^{\text{in}} \tag{4.6}$$

Normalization

First, instance normalization computes batch-wise statistics and whiten features batch wise. The mean and standard deviations are

$$\mu_b = \frac{1}{|\mathcal{C}_b^{\rm in}|} \sum_{\mathbf{k} \in \mathcal{C}_b^{\rm in}} \mathbf{f}_{\mathbf{u},b}^{\rm in}$$
(4.7)

$$\sigma_{bi}^2 = \frac{1}{|\mathcal{C}_b^{\text{in}}|} \sum_{\mathbf{u} \in \mathcal{C}_b^{\text{in}}} (\mathbf{f}_{\mathbf{u},bi}^{\text{in}} - \mu_{bi})^2$$
(4.8)

where $\mathbf{f}_{\mathbf{u},bi}$ indicates the *i*-th channel feature at the coordinate \mathbf{u} with batch index *b*. C_b^{in} is the set of non-zero element coordinates in the *b*-th batch. μ_b indicates the *b*-th batch batch-wise feature mean and σ_{bi} is the *i*-th feature channel standard deviation of the *b*-th batch.

$$\mathbf{f}_{\mathbf{u},bi}^{\text{out}} = \frac{\mathbf{f}_{\mathbf{u},bi}^{\text{in}} - \mu_{bi}}{\sqrt{\sigma_{bi}^2 + \epsilon}}$$
(4.9)

Batch normalization is similar to the instance normalization except that it computes statistics for all batch.

$$\mu = \frac{1}{|\mathcal{C}^{\mathrm{in}}|} \sum_{\mathbf{k} \in \mathcal{C}^{\mathrm{in}}} \mathbf{f}_{\mathbf{u}}^{\mathrm{in}}$$
(4.10)

$$\sigma_i^2 = \frac{1}{|\mathcal{C}^{\mathrm{in}}|} \sum_{\mathbf{u} \in \mathcal{C}^{\mathrm{in}}} (\mathbf{f}_{\mathbf{u},i}^{\mathrm{in}} - \mu_i)^2$$
(4.11)

$$\mathbf{f}_{\mathbf{u},i}^{\text{out}} = \frac{\mathbf{f}_{\mathbf{u},i}^{\text{in}} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$$
(4.12)

Non-linearity Layers

Most of the commonly used non-linearity functions are applied independently element-wise. Thus, an element wise function $f(\cdot)$ can be a rectified-linear function (ReLU), leaky ReLU, ELU, SELU, etc.

$$\mathbf{f}_{\mathbf{u},i}^{\text{out}} = f(\mathbf{f}_{\mathbf{u},i}^{\text{in}}) \text{ for } \mathbf{u} \in \mathcal{C}$$

$$\tag{4.13}$$

4.3 Implementation

A sparse tensor network requires specialized functions to manage the unstructured coordinates of non-zero elements. For example, spatial operations such as convolution and max pooling require 1. generating a new set of coordinates if we use stride > 1 and 2. finding neighbors for all coordinates. In a conventional neural network, random access to arbitrary coordinates requires simply computing the offset from the 0-th element. However, a sparse tensor requires a special data structure to speed

up the random access and searching neighbors. We use a hash table to manage the coordinates of non-zero elements. A hash table allows random access to arbitrary coordinates, but requires extra overhead to compute a hash key and to manage hash key collision.

We delegate all functions associated with creating and searching coordinates to a *coordinate* manager. Internally, the coordinate manager uses a set of hash tables to track the coordinates

4.3.1 Coordinate Manager

A coordinate manager generates a new sparse tensor and finds neighbors among coordinates of nonzero elements. Also, once we create a new set of coordinates, the coordinate manager caches the coordinates and the neighborhood search results these are reused very frequently. For example, in many conventional neural networks, we repeat the same operations in series multiple times such as multiple residual blocks in a ResNet or a DenseNet. Thus, instead of recomputing the same coordinates and same kernel maps, a coordinate manager caches all these results and reuses if it detects the same operation that is in the dictionary is called.

Sparse Tensor Generation

The first step that converts unstructured data into a sparse tensor is discretization. Let continuous coordinates of the data be $C = {\mathbf{x}_i}_{i=1}^N$, the user-defined quantization size be s. We compute the discretized coordinate for all elements in C by dividing the coordinate by s and flooring it to get the integer coordinates $|\mathbf{x}/s|$.

We use a hash table to store the discretized coordinates and use a D-dimensional integer coordinate as the key of the table and use the row index of the coordinate as the value. We use a hash function based on the Murmur hash [1] that sequentially processes all bytes (4 bytes for an integer) of D integers in a coordinate for this hash table.

When there is a coordinate collision, we simply ignore the coordinates that are inserted after the first insertion. However, for 3D perception tasks, we did not see any performance degradation when the quantization size is less than 5cm.

Coordinate Key

Within a coordinate manager, all objects are cached using an unordered map. A coordinate key is a hash key for the unordered map that caches the coordinates of sparse tensors. If two sparse tensors have the same coordinate manager and the same coordinate key, then the coordinates of the sparse tensors are identical and they share the same memory space.

Kernel Map

A kernel map is a pair of coordinate indices that aligns input features to corresponding output features. Specifically, this is an operation that is equivalent to Im2Col in a conventional neural network. We simply iterate over all neighbors defined by $\mathcal{N}(\mathbf{u})$ in Eq. 4.4 to search existing coordinates $\mathcal{N}^{D}(\mathbf{u}) \cap \mathcal{C}^{\text{in}}$. As this search operation can be run independently from each other, we parallelize the kernel map generation with OpenMP [10].



Figure 4.4: Comparison between (top) the convolution on a dense tensor and (bottom) the convolution on a sparse tensor.

4.4 Example Sparse Tensor Network Architectures

We present common sparse tensor network architectures that are widely used for 3D perception in this section. One advantage of generative convolution is that it supports arbitrary and dynamically generated coordinates. Thus it allows us to solve interesting generative tasks such as 3D object completion or 3D shape generation from a feature vector. We visualize generic network architectures, so specific architectures used for each task can be extensions of these proposed models.

4.4.1 Standard Sparse Tensor Networks

Image classification, semantic segmentation, and object detection have been studied widely and the network architecture used for these tasks are variations of basic networks. We present a few 3D sparse tensor networks for these standard tasks. First, a classification network uses a series of strided convolution and pooling to map an input into a feature vector and then to logit scores. A sparse tensor network in Fig. 4.5 also uses a series of strided convolutions and pooling layers to replicate the same architecture on a sparse tensor. We also use residual blocks to replicate the high-dimensional variants of residual networks on a sparse tensor. An example implementation and training of the network is available online.¹

Similarly, we propose a semantic segmentation network in Fig. 4.6 and a single-shot object detection network in Fig. 4.7. These networks are variations of encoder-decoder style architectures widely used for semantic segmentation and sometimes referred to as pyramid networks or U-networks.

¹ "https://github.com/StanfordVL/MinkowskiEngine/tree/master/examples



Figure 4.5: Classification Sparse Tensor Network



Figure 4.6: Semantic Segmentation Sparse Tensor Network



Figure 4.7: Object Detection Sparse Tensor Network

4.4.2 Generative Sparse Tensor Networks

3D generation tasks include completion and reconstruction. First, a shape completion network infers a complete unaltered data or a complete 3D shape from a partial observation or scan of a 3D shape. A 3D reconstruction network also generates 3D shapes as an output, but unlike the completion network, it takes a low-dimensional feature as an input. This feature can be simply a global feature from an image or a one-hot vector indicating a 3D model index.

Both networks use generalized transposed convolution as the key component that upsamples a low-resolution sparse tensor into a high-resolution sparse tensor. The generalized transposed convolution allows the network to dynamically generate new coordinates based on the input, rather than statically generating coordinates beforehand. After the transposed convolution, each non-zero element in the output sparse tensor predicts the likelihood that the current element should be kept or not. The pruning layer then removes the coordinates whose likelihoods are lower than a predefined threshold. We visualize the transposed convolution and pruning pipeline in Fig. 4.8. Note that a generative sparse tensor network consists of multiple blocks of a transposed convolution and pruning pipeline.



Figure 4.8: Upsampling using transposed convolution and pruning

We present a completion network architecture in Fig. 4.9 and a training script.²



Figure 4.9: Completion Sparse Tensor Network

Similarly, a 3D reconstruction network also generates a 3D shape as an output, but unlike the completion network, it generates a 3D shape from a feature vector. We visualize an example network architecture in Fig. 4.10 and a training scrip.³

 $^{^{2} ``}https://github.com/StanfordVL/MinkowskiEngine/blob/master/examples/completion.py$

³ "https://github.com/StanfordVL/MinkowskiEngine/blob/master/examples/reconstruction.py



Figure 4.10: Generative Sparse Tensor Network

Bibliography

- [1] Austin Appleby. Murmurhash. URL https://sites. google. com/site/murmurhash, 2008.
- [2] Christopher Choy, JunYoung Gwak, and Silvio Savarese. 4d spatio-temporal convnets: Minkowski convolutional neural networks. In CVPR, pages 3075–3084, 2019.
- [3] Christopher Choy, Jaesik Park, and Vladlen Koltun. Fully convolutional geometric features. In *ICCV*, 2019.
- [4] Angela Dai, Angel X. Chang, Manolis Savva, Maciej Halber, Thomas Funkhouser, and Matthias Nießner. Scannet: Richly-annotated 3d reconstructions of indoor scenes. In Proc. Computer Vision and Pattern Recognition (CVPR), IEEE, 2017.
- [5] Benjamin Graham. Spatially-sparse convolutional neural networks. *arXiv preprint arXiv:1409.6070*, 2014.
- [6] Benjamin Graham. Spatially-sparse convolutional neural networks. arXiv preprint arXiv:1409.6070, 2014.
- [7] Ben Graham. Sparse 3d convolutional neural networks. British Machine Vision Conference, 2015.
- [8] Benjamin Graham, Martin Engelcke, and Laurens van der Maaten. 3d semantic segmentation with submanifold sparse convolutional networks. CVPR, 2018.
- [9] Baoyuan Liu, Min Wang, Hassan Foroosh, Marshall Tappen, and Marianna Pensky. Sparse convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 806–814, 2015.
- [10] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008.
- [11] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks. ACM SIGARCH Computer Architecture News, 45(2):27–40, 2017.
- [12] Parker Allen Tew. An investigation of sparse tensor formats for tensor libraries. PhD thesis, Massachusetts Institute of Technology, 2016.