Sparse Tensor Collection for Neural Networks

Surya Dutta
Department of Electrical and Computer Engineering
North Carolina State University
Raleigh, USA
sdutta5@ncsu.edu

ABSTRACT

In today's world, sparse tensors are commonly used in deep learning applications as they can reduce the storage and computational requirements of the data that needs to be processed. By utilizing collections of sparse tensors, sparse tensor operations can be extracted and optimized to run faster than dense tensor operations, making them more efficient for certain types of computations. This project deals with the efficient extraction of sparse tensor operations from deep neural network models, in order to optimize and improve model performance for DNNs. VGG-16 and BERT-base Encoder are the two DNN models chosen in this project, and tensor optimizations are performed on them such as pruning and extracting the sparse weights of the layers of the models. Automated Gradual Pruning is implemented and tested on the models. While model optimization and performance evaluation are beyond the scope of this project, this project is an effort towards identifying the sparse layers, analyzing layer operations, and extraction of sparse tensor operations from deep neural network models. Future applications of this project include implementing sparse operations using sparse tensor libraries, or using hardware accelerators, so that model inference and performance can be improved.

KEYWORDS

Sparse Tensor Collection, Deep Neural Networks, Optimization, Pruning

INTRODUCTION

In mathematics and computer science, a tensor is a multi-dimensional array of numerical values that generalize scalars, vectors, and matrices to higher dimensions. A sparse tensor is a tensor in which most of the elements are zero. Sparse tensors are important in many areas of science and engineering where data is often naturally sparse. For example, in image processing, the pixels in an image are often sparse, as only a small fraction of them contain meaningful information. In computational fluid dynamics, the values of a flow field may be zero except in a small subset of the domain where the flow is non-zero. Since sparse tensors contain mostly zero values, they can be more efficiently represented using specialized data structures than dense tensors which contain values for every element.

Sparse tensors are useful in deep learning because they allow for efficient storage and computations by only storing and processing non-zero elements unlike dense tensors. They are also beneficial for handling large datasets, as well as for tasks that involve missing or incomplete data. Additionally, sparse tensors can improve model interpretability and scalability by identifying important features or inputs. In a sparse tensor, the non-zero elements correspond to the most relevant features, which can help researchers and practitioners better understand the relationships between the input and output features of a deep learning model.

Sparse tensor collection is a way to organize and manipulate multiple sparse tensors together as a single data structure. It can be thought of as a higher-level abstraction that allows the efficient manipulation of large, complex datasets. For example, imagine building a recommendation system that considers multiple types of data, such as user demographics, user behavior, and item features. Each of these data types can be represented as a sparse tensor, and a sparse tensor collection can be used to store and manipulate all these tensors together. Sparse tensor collections can also be useful for processing large-scale graph data, where the graph structure can be represented as a collection of sparse adjacency matrices. Sparse tensors, and collections of sparse tensors can be represented using various data structures, depending on the specific implementation and application requirements.

Some of the ways to represent sparse tensors are as follows. First, the coordinate list (COO) format is the simplest and most used representation for sparse tensors, where each non-zero element is stored as a tuple of its indices and value. The COO format is efficient for construction and insertion of new elements, but may require additional processing to perform tensor operations efficiently. Second, the compressed sparse row (CSR) and compressed sparse column (CSC) formats are more efficient for sparse matrix multiplication, which is the focus of this project. In the CSR format, the non-zero elements are stored in a flat array, and the row indices and column indices are stored in separate arrays. The CSC format is similar, but the column indices are stored instead of the row indices. Block compressed row (BCR) formats are useful for representing tensors with large blocks of non-zero elements, where the sparse tensor is partitioned into rectangular blocks, and

each block is represented using a compressed sparse row format. This approach can be more efficient for tensors that contain large blocks of non-zero elements. In the dictionary of keys (DOK) format, non-zero values are stored as key-value pairs in a dictionary, where keys represent indices of the non-zero elements. Hybrid formats are used to represent sparse tensor collections that have a mixture of sparse and dense tensors.

The focus of this project is in the application of sparse tensor collections for neural networks. Sparse tensors are used in neural networks in order to set some of the weights or activations to zero during training or inference. This has the many advantages as follows. Since sparse networks use less memory and fewer computations compared to dense networks, there is improved memory and computational efficiency and this can be beneficial for large models or limited computational resources. Introducing sparsity also improves generalization and reduces overfitting of neural networks by encouraging the model to learn only the most important features and reducing noise in the data. Additionally, sparse networks can be easier to interpret because the nonzero weights or activations correspond to the most relevant features or connections in the network. Also, sparse networks can be compressed more efficiently than dense networks, which can be useful for storage and transfer of models.

Having stated the numerous advantages, sparse tensors can be used in neural networks in different ways, ranging from representing the input data efficiently so that it can be directly fed into the network, or embedding the data from a high-dimensional space to a lower dimensional space, or in regularization techniques, or in the form of sparse layers. Sparse layers can be added to neural networks, or certain layers of an existing network can be made sparse, so that these layers can efficiently perform operations such as matrix multiplication or convolution only on the non-zero values of the input, thereby reducing the number of computations required. In this project, the focus is on introducing sparsity in the layers of a neural network, and then extracting the sparse tensor operations efficiently, so that network training and inference times can be speeded up.

Converting convolution operations to matrix multiplication is a technique used in deep learning to improve the efficiency of convolutional neural networks (CNNs) by taking advantage of the mathematical properties of matrix multiplication. In traditional CNNs, convolution operations are performed by computing the dot product between a small filter and each patch of the input signal/image, sliding the filter over the entire signal/image. This process can be computationally expensive, especially when working with large images or deep networks with many convolution layers. Converting convolution operations to matrix multiplication involves reshaping the filter and input tensor into matrices and performing a matrix multiplication instead of a convolution operation. This technique has several benefits, such as -

1. Increased efficiency: Since matrix multiplication can be performed using highly optimized numerical libraries, this results in faster computation times compared to convolution operations.

2. Parallelization: Multiplication operations can be easily parallelized, allowing for efficient use of GPUs or other parallel computing architectures.

3. Generalization: By representing convolution as a matrix multiplication, it becomes easier to implement a wider range of convolutional operations like dilated convolutions, grouped convolutions, and transposed convolutions.

4. Flexibility: Matrix multiplication allows for the use of different types of convolutions, such as depth-wise separable convolutions, which are more computationally efficient than standard convolutions.

RELATED WORK

This project uses concepts from [1], which proposes a novel architecture for performing efficient sparse tensor operations in deep learning applications. Specifically, this research is about the development of a new sparse tensor multiplication algorithm that utilizes the sparsity pattern of both operands simultaneously, resulting in significant performance improvements over existing algorithms. The Dual Side Sparse Tensor Core (DSSTC) algorithm takes advantage of the sparsity patterns of both the left and right operands, using different techniques to optimize the computation on each side. The algorithm consists of two stages, where in the first stage, the algorithm computes the "non-zero tiles" of the left operand, which are the tiles that contain at least one non-zero element. In the second stage, the algorithm computes the product of the non-zero tiles of the left operand with the non-zero elements of the right operand. This paper discusses the two types of sparsity in DNN models, namely the sparsity in weights and the sparsity in activations. Since sparsity in activations is quite difficult to exploit owing to its dynamic and unpredictable nature, this project only deals with inducing sparsity in the weights and then extracting their sparse tensor operations.

The networks that were considered by [1] include VGG-16, ResNet-18, Mask R-CNN, and BERT-base encoder. The networks that were considered in this project are VGG-16 and the BERT-base transformer model.

Models	Pruning Scheme	Dataset	Accuracy
VGG-16	AGP [73]	ImageNet	88.86% (top 5)
ResNet-18		ImageNet	86.46% (top 5)
Mask R-CNN		COCO	35.2 (AP)
BERT-base encoder	MP [30] [54]	SQuAD	83.3 (F1 score)
RNN	AGP	WikiText-2	85.7 (ppl)

Table 1: Details of the DNN models

A.**VGG-16:** The VGG-16 is a convolutional neural network architecture that was introduced in 2014 by a team of researchers at the Visual Geometry Group (VGG) of the University of Oxford. It is a deep neural network consisting of 16 convolutional and fully connected layers and is known for its exceptional performance on image classification tasks. The VGG-16 architecture consists of a series of convolutional layers, followed by max pooling layers, and finally fully connected layers. Each convolutional layer applies a set of learned filters to the input image, which extract different features from the image. The max pooling layers then reduce the dimensionality of the output by taking the maximum value of each local region, further reducing the number of parameters in the model.

The fully connected layers at the end of the network perform the classification task, mapping the extracted features to the corresponding output classes. The architecture of VGG-16 is such that it uses small filters with a fixed size of 3x3, and the number of filters is doubled after each pooling layer, resulting in a large number of parameters. VGG-16 has achieved state-of-the-art performance on several benchmark image classification tasks. However, despite its impressive performance, VGG-16 has some limitations. Its large number of parameters makes it computationally expensive and memory-intensive to train, and its fixed filter size limits its ability to capture larger-scale features in the input images. Therefore, subsequent architectures such as ResNet-18 and Mask R-CNN are also explored in [1].

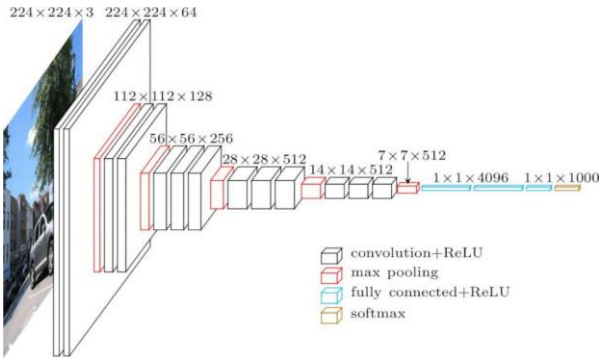

Figure 1: VGG-16 architecture

B. **BERT-base Encoder:** The BERT-base encoder [3] is a neural network model that was introduced by Google researchers in 2018. BERT was designed to pre-train a deep bidirectional representation of the natural language text, which can be fine-tuned on specific downstream natural language processing (NLP) tasks. The BERT base encoder architecture consists of a stack of 12 transformer blocks, each containing multi-head self-attention and feedforward layers. The transformer blocks are connected to each other through residual connections and layer normalization, allowing the model to learn complex patterns. The BERT-base model is pre-trained on a large corpus of text data using two tasks, namely a masked language modeling task and a next sentence prediction task. In the masked language modeling task, a certain percentage of tokens in a sentence are masked and the model is trained to predict the masked tokens based on the surrounding context. In the next sentence prediction task, the model is trained to predict whether two given sentences are consecutive or not. Once pre-trained, the BERT base encoder can be fine-tuned on different NLP applications such as sentiment analysis, named entity recognition, question-answering, etc. BERT-base uncased is a variant of the BERT model. The "uncased" in its name refers to the fact that all text in the model is in lowercase and there is no information about the original capitalization of the text. The model has 110 million parameters, and it has achieved state-of-the-art results on many NLP tasks.

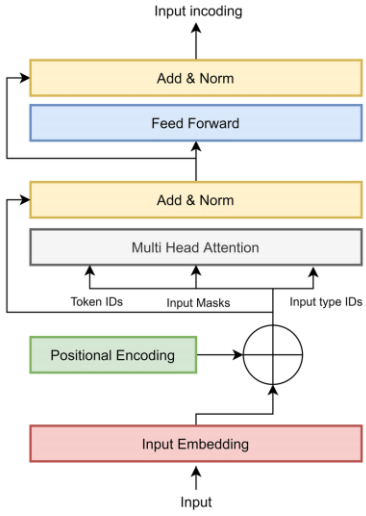

Figure 2: BERT-Base Encoder architecture

[2] discusses pruning techniques that were used for the networks in [1], particularly for the VGG-16 model. The paper investigates the efficacy of model pruning, which is a popular technique for model compression, and explores the factors that affect its effectiveness. Pruning involves

removing a subset of weights or neurons from a trained neural network to reduce its size and computational complexity. Model pruning is a widely used technique for model compression, as it can reduce the size of a model without significantly sacrificing its performance. The effectiveness of pruning varies depending on the specific model and the level of pruning. There are several types of pruning techniques such as:

1. Weight pruning, which involves pruning individual weights from the network that have the least impact on the network's output. This can be done in different ways, such as setting small weights to zero or removing them entirely. This project incorporates only this type of pruning.

2. Neuron pruning, where entire neurons in the network are removed if they are determined to be redundant or unnecessary for the network's output. This can be done by measuring the impact of each neuron on the output and removing the ones with the least impact.

3. Filter pruning, which involves removing entire convolutional filters from the network that have the least impact on the network's output. This can be done by measuring the impact of each filter on the output and removing the ones with the least impact.

4. Structured pruning, which deals with removing entire groups of weights or filters from the network, rather than pruning individual weights or filters. This can be done in different ways, such as removing entire rows or columns of weight matrices or entire channels of filters.

5. Channel pruning, where entire channels of filters are removed based on their importance to the network's accuracy.

METHODOLOGY

Once the network of interest is chosen, the network is trained, and then sparsity in introduced into the network at the weights of select layers. The network that was chosen was VGG-16. Once the network is trained, the convolution operations are converted into matrix multiplication, as this improves efficiency, and also due to the fact that since most accelerators (GPUs or tensor cores) rely on matrix-multiply primitives, it becomes easier for the accelerator to further speedup the model or network when the convolution operations are converted to matrix multiplication primitives.

This is done using the im2col algorithm. The im2col algorithm works by taking a sliding window over the input image and stacking the image patches into columns. The size of the sliding window is determined by the size of the convolutional filter used in the network. For example, in a 3x3 convolutional filter, the sliding window would be a 3x3 square that moves across the image, one pixel at a time. As the sliding window moves across the input image, the pixels within the window are flattened into a column vector, and each of these column vectors is then stacked horizontally to form a matrix. This matrix represents the input image in a format that can be multiplied by the convolutional filter. The weights of the convolutional filter are also flattened and converted into a row vector, and the dot product of the input matrix and the filter weights row vector is calculated to produce a single output value. By using the im2col algorithm to convert the input image into a matrix, the convolution operation can be performed efficiently using standard matrix multiplication operations. This can significantly speed up the computation of the convolution operation, especially in neural networks where the size of the input image and the number of filters is very large. Since the im2col algorithm re-arranges and expands the convolution's input feature maps into a matrix, called lowered feature map, whose each row corresponds to a location of 2-dim sliding window in convolution's input feature maps, the im2col on weight parameters is nothing but flattening each K×K×C kernel.

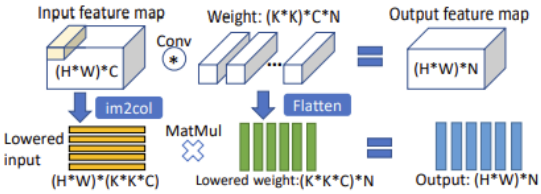

Fig. 1: The im2col-based transformation of CONV to GEMM.

Figure 3: im2col-based transformation

Libraries like the NVIDIA CuDNN (CUDA Deep Neural Network) library provides a collection of primitives for deep neural networks, including convolution, pooling, normalization, activation functions, and tensor transformation operations. This library is designed to optimize the performance of deep learning applications by providing low-level GPU acceleration for key operations used in neural networks, and hence it automatically converts convolution operations to matrix-multiply primitives. Since this is not open-source, this project could not utilize this library.

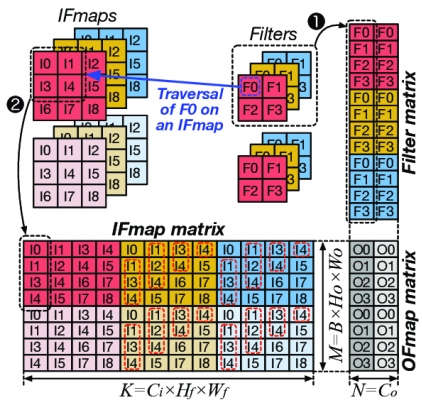

Figure 4: Conversion of convolution to matrix multiplication using im2col

After converting convolution operations to matrix-multiply primitives for each convolutional layer of the neural network, the weights of each layer were extracted, and then the next step was to apply pruning to the network. This project made use of the same pruning technique used for VGG-16 in [1], which is the automated gradual pruning algorithm.

Automated Gradual Pruning (AGP) aims to remove a large number of redundant or unnecessary connections in the neural network, while maintaining the overall accuracy of the model. This method uses a gradual pruning approach, where connections are pruned in small increments over time, rather than all at once, to minimize the risk of over-pruning and losing too much accuracy. The AGP method works by assigning a "score" to each connection in the network based on its importance to the overall performance of the model. Connections with low scores are then pruned in small increments over a period of time iteratively, while the model is retrained to maintain its accuracy. This process is repeated until the desired level of pruning is achieved. Thus, AGP performs very well as a pruning method for neural networks that can be used to reduce their computational complexity and memory requirements, without sacrificing accuracy. One of the key advantages of AGP is that it is an automated method, meaning that it does not require manual tuning or selection of hyperparameters. Instead, it uses a "layer-wise" pruning approach, where each layer in the network is pruned separately, and the pruning rate for each layer is automatically determined based on the importance of the connections in that layer. Only the initial pruning rate and the desired pruning rate (at the end of training) need to be defined by the user. Another advantage of AGP is that it can be applied to a wide range of deep neural network architectures.

```
def AGP(model, x_train, y_train, x_test, y_test, epochs):
    initial_sparsity = 0.0
    final_sparsity = 0.5
    sparsity_delta = (final_sparsity - initial_sparsity) / epochs

    # Define pruning schedule
    pruning_schedule = tfmot.sparsity.keras.PolynomialDecay(
        initial_sparsity=initial_sparsity,
        final_sparsity=final_sparsity,
        begin_step=0,
        end_step=epochs,
    )

    # Define model for pruning
    pruned_model = tfmot.sparsity.keras.prune_low_magnitude(model, pruning_schedule=pruning_schedule)

    pruned_model.compile(
        loss='categorical_crossentropy',
        optimizer='adam',
        metrics=['accuracy']
    )

    # Fit the pruned model
    pruned_model.fit(
        x_train,
        y_train,
        epochs=epochs,
        validation_data=(x_test,y_test),
    )

    # Strip pruning information from model
    final_model = tfmot.sparsity.keras.strip_pruning(pruned_model)

    # Evaluate the model on test data
    _, accuracy = final_model.evaluate(x_test, y_test, verbose=0)
    print('Test accuracy after pruning:', accuracy)

    return final_model
```

Figure 5: AGP Workflow

In case of the BERT-base encoder, movement pruning is used. Movement pruning is based on the idea of pruning connections in the network based on their "movement", or how much they change during the training process. Connections with low movement scores, indicating that they are relatively stable and not contributing much to the overall performance of the model, are pruned from the network. One advantage of this type of pruning is that it is a dynamic pruning method, meaning that connections are pruned during training rather than after the network has been trained. This allows the network to adapt to changing data and learn more efficiently over time. However, movement pruning is computationally expensive, as it requires tracking the movement of each connection in the network during training. This was not implemented.

RESULTS & ANALYSIS

Figure 6 demonstrates the conversion of convolution operations into matrix-multiplication primitives on a pretrained VGG-16 CNN model, which was trained on the ImageNet database. A random input tensor was created to test the convolution operation, and the output tensors of the convolution and matrix multiplication operations were compared.

```
Accelerating Deep Learning (CSC591) Project
Surya Dutta (SID:200481187)

as import VGG16 and extract the weights
## Convert the convolution operation to matrix multiplication using im2col

[ ] import torch
    import numpy as np

    # Load the pre-trained VGG16 model
    vgg16 = torch.hub.load('pytorch/vision', 'vgg16', pretrained=True)

    # Extract the weights and bias of the conv layer
    conv_layer = vgg16.features[0]
    conv_weights = conv_layer.weight.data.numpy()
    conv_bias = conv_layer.bias.data.numpy()

    # Convert the weights to matrix multiplication format using im2col
    kernel_size = conv_weights.shape[2]
    input_channels = conv_weights.shape[1]
    output_channels = conv_weights.shape[0]
    conv_weights_im2col = conv_weights.reshape((output_channels, -1))
    conv_weights_im2col = torch.tensor(conv_weights_im2col, dtype=torch.float32)

    # Create a random input tensor to test the convolution operation
    input_data = np.random.rand(1, 224, 224, 3)
    input_tensor = torch.from_numpy(input_data).permute(0, 3, 1, 2).float()

    # Convert the input tensor to matrix multiplication format using im2col
    input_data_im2col = torch.nn.functional.unfold(input_tensor, kernel_size=kernel_size, stride=1, padding=1)

    # Compute the convolution operation using matrix multiplication
    output_im2col = torch.mm(conv_weights_im2col, input_data_im2col.view(input_channels*kernel_size**2, -1)) + conv_bias.reshape(-1, 1)
```

Figure 6: Implementation of the im2col algorithm

It was observed that the comparison yielded a “false” Boolean value, and this is because whenever convolution is converted to matrix multiplication, the output features of the convolution are embedded in the matrix multiplication result tensor. This can be observed in Figure 7 & Figure 8.

```
[ ] output_im2col

tensor([[[[ 3.4280e-01,  4.3956e-01,  7.9315e-01,  ..., -4.6656e-01,
            5.8460e-01,  1.0061e+00],
          [ 1.6914e-01,  1.1756e-02,  2.7992e-01,  ..., -1.1449e-01,
           -4.3815e-01,  4.4694e-01],
          [ 4.7669e-01, -6.6052e-01,  2.0781e-01,  ...,  7.8127e-01,
           -5.5718e-02, -2.3059e-01],
          ...,
          [ 5.8532e-01,  3.4742e-01,  2.1916e-01,  ..., -8.7552e-01,
            1.3979e+00,  6.4843e-01],
          [ 7.2585e-01,  1.2130e+00,  3.8063e-01,  ...,  2.0809e-01,
            1.0244e+00,  4.6748e-01],
          [ 5.2119e-01,  1.5214e+00,  7.2042e-01,  ...,  5.2138e-01,
            8.1091e-01,  3.0457e-01]],

         [[ 1.5425e+00,  9.5309e-01,  2.3355e-01,  ...,  8.1167e-01,
            1.0292e+00, -5.2229e-01],
          [ 1.3471e+00,  4.4335e-01,  3.7626e-01,  ..., -5.6005e-01,
            3.2387e-01, -6.3802e-01],
          [ 1.3303e+00,  1.2965e+00,  8.1795e-01,  ..., -2.5257e-01,
            3.9713e-01, -1.1236e+00],
          ...,
```

Figure 7: Matrix mutliplication result

```
output_conv

tensor([[[[ 3.4280e-01,  4.3956e-01,  7.9315e-01,  ..., -4.6656e-01,
            5.8460e-01,  1.0061e+00],
          [ 1.6914e-01,  1.1756e-02,  2.7992e-01,  ..., -1.1449e-01,
           -4.3815e-01,  4.4694e-01],
          [ 4.7669e-01, -6.6052e-01,  2.0781e-01,  ...,  7.8127e-01,
           -5.5717e-02, -2.3059e-01],
          ...,
          [ 5.8532e-01,  3.4742e-01,  2.1916e-01,  ..., -8.7552e-01,
            1.3979e+00,  6.4843e-01],
          [ 7.2585e-01,  1.2130e+00,  3.8063e-01,  ...,  2.0809e-01,
            1.0244e+00,  4.6748e-01],
          [ 5.2119e-01,  1.5214e+00,  7.2042e-01,  ...,  5.2138e-01,
            8.1091e-01,  3.0457e-01]],

         [[ 1.5425e+00,  9.5309e-01,  2.3355e-01,  ...,  8.1167e-01,
            1.0292e+00, -5.2229e-01],
          [ 1.3471e+00,  4.4335e-01,  3.7626e-01,  ..., -5.6005e-01,
            3.2387e-01, -6.3802e-01],
          [ 1.3303e+00,  1.2965e+00,  8.1795e-01,  ..., -2.5257e-01,
            3.9713e-01, -1.1236e+00],
```

Figure 8: Convolution operation result

After this, once the convolution operations of the layers were converted to matrix multiplication primitives, L1-unstructured pruning, as well as the AGP algorithm were tested on the network. For the AGP algorithm, the dataset which was used to train the VGG-16 model was the MNIST dataset. Figure 9 and Figure 10 show the implementation of L1-unstructured pruning and AGP.

```
WARNING:absl:`lr` is deprecated in Keras op
Epoch 1/5
/usr/local/lib/python3.10/dist-packages/ten
  return dispatch_target(*args, **kwargs)
1875/1875 [==============================]
  return dispatch_target(*args, **kwargs)
1875/1875 [==============================]
Epoch 2/5
1875/1875 [==============================]
Epoch 3/5
1875/1875 [==============================]
Epoch 4/5
1875/1875 [==============================]
Epoch 5/5
1875/1875 [==============================]
Test loss: 0.0
Test accuracy: 0.11349999904632568
```

Figure 9: AGP implementation

```
prune.l1_unstructured(module, name="bias", amount=3)

Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))

[ ] print(list(module.named_parameters()))
    print(list(module.named_buffers()))
    print(module.bias)
    print(module._forward_pre_hooks)

    [('weight_orig', Parameter containing:
    tensor([[[[-5.5373e-01,  1.4270e-01,  5.2896e-01],
              [-5.8312e-01,  3.5655e-01,  7.6566e-01],
              [-6.9022e-01, -4.8019e-02,  4.8409e-01]],

             [[ 1.7548e-01,  9.8630e-03, -8.1413e-02],
              [ 4.4089e-02, -7.0323e-02, -2.6035e-01],
              [ 1.3239e-01, -1.7279e-01, -1.3226e-01]],

             [[ 3.1303e-01, -1.6591e-01, -4.2752e-01],
              [ 4.7519e-01, -8.2677e-02, -4.8700e-01],
              [ 6.3203e-01,  1.9308e-02, -2.7753e-01]]],
```

Figure 10: L1-unstructured pruning

CONCLUSIONS & FUTURE WORK

In conclusion, leveraging sparsity in neural network models is one of the efficient ways to accelerate DNN architectures. This project focused on introducing sparsity in deep neural networks, identifying the sparse layers, and extracting the tensor operations so that they could be productively utilized in order to improve and optimize model performance and inference. This project could possibly serve as a precursor to further projects in the domain of acceleration of sparse neural networks. The code for this project is attached with this submission.

ACKNOWLEDGMENTS

I would like to thank Dr. Jiajia Li and Mr. Yavuz Tozlu for their valuable guidance and feedback throughout the course of this project.

REFERENCES

[1] Wang, Y., Zhang, C., Xie, Z., Guo, C., Liu, Y., & Leng, J. (2021, June). Dual-side sparse tensor core. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)* (pp. 1083-1095). IEEE.

[2] Zhu, Michael, and Suyog Gupta. "To prune, or not to prune: exploring the efficacy of pruning for model compression." *arXiv preprint arXiv:1710.01878* (2017).

[3] Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

[4] https://h-huang.github.io/tutorials/intermediate/pruning_tutorial.html

[5] https://huggingface.co/bert-base-uncased

[6] https://nvidia.github.io/MinkowskiEngine/sparse_tensor_network.html

[7] https://medium.com/huggingface/is-the-future-of-neural-networks-sparse-an-introduction-1-n-d03923ecbd70

[8] https://ai.stackexchange.com/questions/11172/how-can-the-convolution-operation-be-implemented-as-a-matrix-multiplication

[9] https://pytorch.org/cppdocs/api/function_namespaceat_1a868365503b5baa36722dad3c6ded707a.html

[10] Kumar Chellapilla, Sidd Puri, Patrice Simard. High Performance Convolutional Neural Networks for Document Processing. Tenth International Workshop on Frontiers in Handwriting Recognition, Université de Rennes 1, Oct 2006, La Baule (France). ffinria-00112631

[11] https://developer.nvidia.com/cudnn