
STen: An Interface for Efficient Sparsity in PyTorch

Andrei Ivanov

Nikoli Dryden

Torsten Hoefer

Department of Computer Science, ETH Zürich
{firstname.lastname}@inf.ethz.ch

Abstract

As deep learning models grow, sparsity is becoming an increasingly critical component of deep neural networks, enabling improved performance and reduced storage. However, existing frameworks offer poor support for sparsity. They primarily focus on sparse tensors in classical formats such as COO and CSR, which are not well suited to the sparsity regimes typical of deep learning, and neglect the broader sparsification pipeline necessary for using sparse models. To address this, we propose a new sparsity interface for PyTorch, STen, that incorporates sparsity layouts for tensors (including parameters and transients, e.g., activations), sparsity-aware operators, and sparsifiers, which define how a tensor is sparsified, and supports virtually all sparsification methods. STen can enable better sparse performance and simplify building sparse models, helping to make sparsity easily accessible.

1 Introduction

Deep learning models are growing voraciously, and require ever greater amounts of compute and memory [10, 15, 18, 20]. To address this, sparsity has emerged as a major research and engineering direction [6, 9]. Sparsity is widely used to reduce storage requirements and improve performance during inference. More recent work has begun to focus on sparsity during training, which can also improve performance while helping to break the memory wall for large models. Indeed, frameworks have begun to provide direct support for sparse tensors. The PyTorch [19] `torch.sparse` module includes COO and CSR tensors and a limited set of operations and the TensorFlow [1] `tf.sparse` module similarly supports COO tensors. However, these modules primarily support sparse tensors, rather than an entire sparsification pipeline, and lack native support for sparsification operators that can efficiently produce sparse tensors. They therefore offer limited productivity improvements and do not provide a clear path toward supporting broad usage of sparsity. Further, at sparsities common in machine learning (50–95%), although classical sparse matrix formats reduce storage, they perform worse than dense implementations. While blocked formats (e.g., ELL, BCSR) support efficient implementations by calling dense kernels for each block, they restrict where nonzeros can be placed and can limit the information preserved after sparsification. Another approach is to use masks, which emulate sparsity by zeroing out elements, but offer no storage reduction.

To address this, we first propose a new programming model for sparsity in PyTorch (§2), overviewed in Fig. 1. This model consists of three components: *sparsity layouts* for tensors; *operators*, which provide implementations for computations with any combination of sparsities for input and output tensors; and *sparsifiers*, which are applied to operator outputs to compute a new sparse tensor. Sparsifiers are further classified as *streaming*, *blocking*, or *materializing*, based on the number of output values they require. Our model supports the vast majority of sparsification approaches, and enables them to be implemented efficiently; for example, threshold pruning is a streaming sparsifier, and a high-performance implementation could be inlined into operators. We provide an initial implementation of this model, STen, in PyTorch (§3) for CPU inference, but aim to lay the foundation for sparse training on accelerators. We also provide performant operators and an $n:m$ sparsity format [16, 27], where each group of m elements has n nonzeros, which is a middle ground between

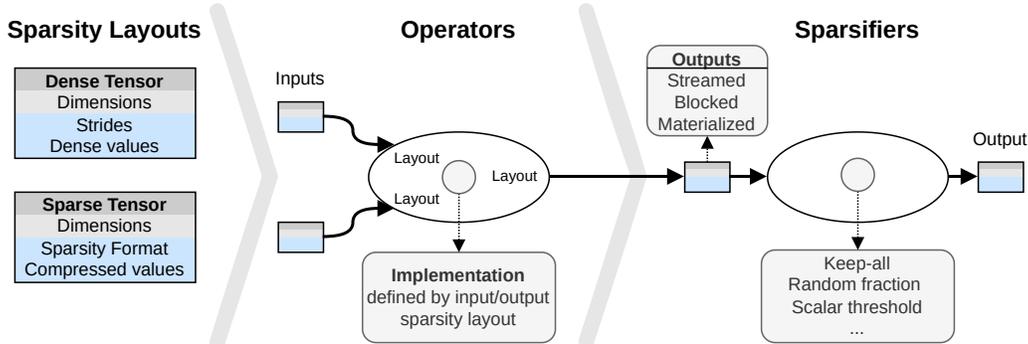


Figure 1: Overview of our proposed sparsity programming model.

unstructured and block sparsity that performs well for deep learning workloads and is practical on CPUs and GPUs. With STen, we show (§4) that sparse-dense matrix multiplication can outperform PyTorch’s built-in sparsity in the 50–90% sparsity regime: up to $54\times$ faster than its COO format and $3\times$ faster than its CSR format. We also match or outperform dense formats at 60% or greater sparsity. Our code and examples are available at <https://github.com/spcl/sten>.

2 Programming Model for Sparsity

We now introduce our programming model for sparsity. It consists of three concepts, sparsity layouts, operators, and sparsifiers, which we discuss in turn. We discuss implementation details in §3. While existing frameworks have some support for sparse tensors and operators, they lack explicit support for sparsifiers, which are manually incorporated by users in an ad hoc manner. Instead, we manage the entire sparsification pipeline, simplifying usage while enabling improved performance.

Sparsity layouts define the sparsity structure used by a tensor, extending the usual memory layout by annotating the sparsity format (e.g., CSR, COO, $n:m$) and associated parameters (e.g., n and m).

Operators are any standard operator, with any number of input and output tensors. Each tensor may have any memory and sparsity layout, and an operator may have different implementations for each combination of tensor layouts for maximum performance.

Sparsifiers define how to decide which output values to keep, and are associated with each output of an operator. A sparsifier can be thought of as a special kind of operator, and may include additional inputs that delay its application until they are ready (e.g., gradients for first-order sparsification). Note a sparsifier may produce output in a different sparsity layout than what its associated operator outputs. We use the term *sparse operator* to refer to the combination of an operator and sparsifier.

We further classify sparsifiers as one of *streaming*, *blocking*, or *materializing*. A streaming sparsifier is applied to each output value to decide whether to drop it, in a single pass, before writing to the output tensor. Blocking sparsifiers require a small set of output values to decide which ones to drop. Finally, materializing sparsifiers require the operator to fully store all values. Note that a sparsifier may be fused into an operator for performance, or used standalone to convert dense tensors to sparse.

Table 1 lists example sparsifiers and their characteristics. The trivial *keep-all* sparsifier preserves all produced values and is the default for dense tensors. It is not limited to dense tensors, however: the sum of two sparse tensors with a keep-all sparsifier may produce a new sparse tensor with nonzero values given by the union of the nonzeros of the inputs. A *random fraction* sparsifier drops values with a fixed probability, while a *scalar threshold* drops them if they are less than a fixed threshold. *Scalar fraction* drops the smallest portion of the values (i.e., magnitude pruning) and *block-wise fraction* drops entire blocks with the smallest combined absolute magnitude. *Per-block fraction* drops the smallest proportion of values within fixed blocks of elements. Finally, more advanced *complex weight sparsifiers*, which require additional information such as the loss or gradients, are also supported. These examples are not exhaustive, and our model can support nearly any sparsifier.

2.1 Constructing Sparse Models

We consider two cases for building sparse models: constructing a model from scratch or sparsifying an existing model. A sparse model is set up by providing a list of tensors and a desired sparsity layout

Table 1: Sparsifier types and examples, the number of passes over a tensor made, their memory requirements (nnz total nonzeros, block size b when blocking), and sparsifier type. Some complex weight sparsifiers could be implemented more efficiently than with materialization.

Sparsifier	Examples	Passes	Memory	Type
Keep-all	Sparse add	1	$\mathcal{O}(1)$	streaming
Random fraction	Dropout [22]	1	$\mathcal{O}(1)$	streaming
Scalar threshold	ReLU [14]	1	$\mathcal{O}(1)$	streaming
Scalar fraction	Magnitude [5, 28]	2	$\mathcal{O}(nnz)$	materializing
Block-wise fraction	Block magnitude [12]	2	$\mathcal{O}(nnz)$	materializing
Per-block fraction	$n:m$ [16, 27]	2	$\mathcal{O}(b)$	blocking
Complex weight sparsifiers	Movement [21], ℓ_0 [13], etc. [9]	≥ 1	$\mathcal{O}(nnz)$	materializing

for each. This provides enough information for a library to initialize tensors and dispatch operators to specific sparse formats. If an operator implementation is not available, the user can provide one or convert tensors to a supported sparsity layout. Note that all tensors are used as operator inputs, outputs, or both; for simplicity, we will ignore sparsity for model inputs and outputs. The remaining input-only tensors are typically *weights*. All other tensors are used as both inputs and outputs and occur within the computation graph (e.g., activations); we refer to them as *intermediate tensors*. Note that in practice, intermediate tensors do not exist until runtime in most frameworks, so their sparsity layout is instead defined by the operator that produces them.

Constructing a sparse model from scratch is similar to the typical process in PyTorch, but tensors and operators in its computational graph are annotated with specific sparsity layouts.

Sparsifying an existing model requires marking a subset of the model’s tensors as sparse. While this is straightforward for weights, making intermediate tensors sparse is more challenging. Unlike when building a model from scratch, we cannot mark operators as sparse: this would require modifying or rewriting the original model definition, a significant overhead for a user. Further, identifying all operators in a model is hard (not all have names or are registered, e.g., `nn.functional` operators). Instead, to identify intermediate tensors, we can run the model once to collect this information.

We now discuss sparse inference, then proceed to training. To sparsify existing dense weights or load sparse weights, we need only the desired sparsity layout and sparsifier. If a non-materializing sparsifier is given, we first trivially convert it to a materializing version. Then the weights are sparsified and subsequent operator calls will use the sparse version. Intermediate tensors are sparsified at runtime, as they do not exist in advance.

For training, we need to also consider error signals (sometimes called neural gradients) and gradients, which may have independent sparse layouts and sparsifiers from their associated forward pass tensors. These are treated identically to intermediate tensors in the forward pass. However, note that in this case, weight tensors are no longer input-only, as gradient updates are applied to them. This is not a significant change from the user perspective, and mainly implies that materializing sparsifiers may be less efficient and sparsifying on the fly with the gradient update operator may be faster.

3 STen Implementation

We now discuss the implementation of our sparsity programming model in PyTorch, STen. The interface is in Python and interoperates with standard PyTorch models to enable a user to construct a sparse model from scratch or sparsify an existing dense model.

Model construction. To build a sparse model from scratch, the user essentially constructs a model as usual in PyTorch, but uses sparse versions of operators and tensors. The STen API supports sparse tensor and parameter classes, which can store tensors using any sparsity layout, and can be extended by the user to support additional layouts. Operators (e.g., matrix multiplication) can infer the layout of their input tensors, but the desired output tensor layout must be explicitly specified. Finally, where desired, sparsifiers can be associated with an operator to select which output values to keep.

```
import torch, sten
a, b = torch.randn(10, 20), torch.randn(10, 20)
```

```

sparse_add = sten.sparsified_op(orig_op=torch.add,
    out_fmt=[(sten.KeepAll(), torch.Tensor,
               sten.RandomFractionSparsifier(0.5), sten.CsrTensor)],
    grad_out_fmt=[(sten.KeepAll(), torch.Tensor,
                    sten.RandomFractionSparsifier(0.5), sten.CsrTensor)])
c = sparse_add(a, b) # dense + dense -> sparse

```

Sparsifying existing models. Alternatively, one can construct a sparse model from an existing dense model. To do this, the user provides the original PyTorch model, a list of tensor identifiers belonging to the model, and the desired sparsity layout and sparsifier for each tensor. To identify weight tensors, we use the fully-qualified names provided by PyTorch. For intermediate tensors, we obtain names using the `torch.fx` tracing framework. Note that adding sparsity to an intermediate tensor is equivalent to using a sparse operator with that output layout. Sparsifiers are then applied to existing dense weight tensors, which are then replaced by a sparse tensor, and operators that output sparse intermediate tensors are replaced with the corresponding sparse operator.

```

model = torch.nn.TransformerEncoderLayer(d_model=512, nhead=8)
sb = sten.SparsityBuilder(model)
sb.set_weight(name='linear1.weight', out_format=sten.CooTensor,
              initial_sparsifier=sten.ScalarFractionSparsifier(0.5))
sb.set_interm(name='relu', out_format=sten.CooTensor,
              external_sparsifier=sten.ScalarFractionSparsifier(0.5))
sparse_model = sb.get_sparse_model()

```

Identifying intermediate tensors uniquely is challenging in the current PyTorch API, as the tensors are materialized at runtime and only accessible via tracing. We found three approaches for this: `torch.fx`, exporting to ONNX [17], or low-level TorchScript tracing. However, ONNX operators do not map perfectly to PyTorch operators, making it difficult to recover the original model. TorchScript tracing, while powerful and flexible, introduces significant complexity. We therefore use `torch.fx` as replacing tensors and operators with it is simple.

Operator implementations. Additionally, providing operator implementations for all combinations of input and output sparsity layouts is unrealistic given the large number of combinations, especially when users may provide their own formats. To overcome this, we support an operator extension API that accepts the operator name, a matching pattern for supported input and output sparsity layout combinations, and an actual implementation. When such an operator is called with matching tensors, we dispatch to the given implementation, which can make use of existing high-performance implementations (e.g., [2, 4, 7, 8]). If an implementation does not exist, the interface can optionally convert input tensors to a supported format; however, this requires the destination format to support this transformation losslessly (e.g., CSR) and sacrifices many opportunities for performance with the original layout (e.g., optimizations for blocked layouts). Otherwise, we fall back to using a dense implementation and issue a warning. This enables incremental optimization by performance engineers, who can decide which operators it is most profitable to provide sparse implementations for. It also allows users to explore the effects of sparsity on models (albeit with a performance penalty).

```

@sten.register_fwd_op_impl(operator=torch.add,
    inp=(sten.CsrTensor, sten.CooTensor, None, None),
    out=[(sten.RandomFractionSparsifier, sten.CooTensor)])
def sparse_add_fwd_impl(ctx, inputs, output_sparsifiers):
    input, other, alpha, out = inputs
    [out_sp] = output_sparsifiers
    return native_cpp_impl(input, other, alpha=alpha, out=out, sp=out_sp)

```

Backpropagation. Our programming model also supports sparsity during backpropagation. However, supporting automatic differentiation and backpropagation on arbitrary sparsity layouts and operators is challenging in PyTorch, as its autograd engine makes assumptions that limit its flexibility. In particular, the autograd engine is implemented in C++ and expects instances of `torch.Tensor`, meaning we cannot provide a separate class or wrap existing tensors to support additional sparsity layouts, and also limits the usefulness of custom implementations via `torch.autograd.Function`. Further, while `torch.Tensor` can be subclassed, these subclasses will still inherit the dense memory of the base class. To address this, we take a hybrid approach, and inherit from the empty tensor, wrapping custom

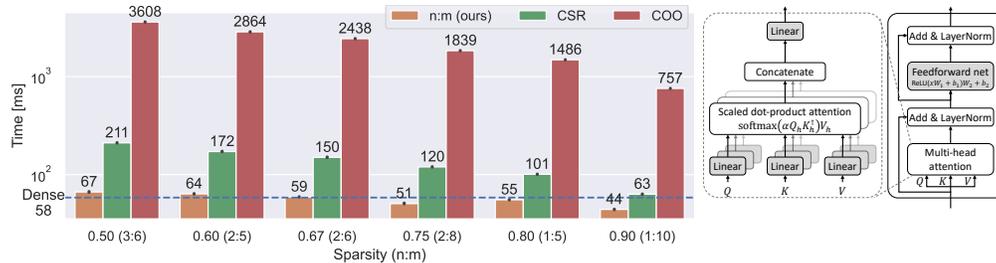


Figure 2: **Left:** Forward-pass runtime of a single BERT_{BASE} [3] encoder layer from HuggingFace [24]. We sparsify the weights of feedforward layers and attention projections (except biases; shaded **right**).

sparsity layouts inside of this, and then override the forward and backward functions of operators that access the tensor to use the wrapped sparse data. We also intercept gradient assignments to make sure that sparse data is not lost. This allows STen to easily support new sparse layouts and operators.

4 Case Studies

We now evaluate our sparsity programming model and the STen implementation, first by considering two case studies showing its productivity, and then examining its performance.

Productivity. We first consider sparsifying an existing BERT_{BASE} [3] encoder layer from HuggingFace [24]. Sparsifying all linear layer weights as well as the GELU activation output requires only about ten lines of code; this would not otherwise be possible in PyTorch without rewriting the model from scratch or manually replacing operators and tensors in the same manner as STen. We then consider constructing a simple sparse MLP from scratch. While both STen and `torch.sparse` can accomplish this with comparable complexity, STen is significantly more flexible, and supports arbitrary sparsity layouts and operators without modifying PyTorch (§3). For full examples, see §A.

Performance. We now consider the performance of the forward pass (i.e. for training) of a BERT_{BASE} encoder layer, which is a popular target for sparsification (e.g., [11, 21, 26]), and applicable to both training and inference environments. We use the BERT_{BASE} (uncased) model from HuggingFace [24] with batch size 8 and sequence length 128, and apply a random fraction sparsifier to sparsify weights. We run experiments on an Intel i7-4770 CPU. We compare the default PyTorch (v1.11) dense implementation, `torch.sparse` using CSR and COO formats, and our own *n:m* sparse layout. Our *n:m* layout is designed for high speed at sparsities common in deep learning. It uses accumulation in vector registers to perform efficient multiplications on CPU, following OpenBLAS [23, 25]. The *n:m* blocks are aligned along the non-contraction axis and put in equally-sized groups with the same nonzero pattern to avoid branch mispredictions; original locations are stored in dense integer tensors.

We show results in Fig. 2. Our sparsity implementation significantly outperforms `torch.sparse`, and is up to 54× faster than its COO format and 3× faster than its CSR format. It also matches or outperforms dense PyTorch tensors when at least 60% sparse, while saving significant memory.

5 Discussion

We have proposed, and conducted initial evaluations on, a new interface for sparsity in PyTorch, STen. By directly incorporating the notion of a sparsifier into the programming model, we allow the interface to both better optimize performance (e.g., by not materializing tensors before sparsification and fusing sparsification into operators) and provide a complete pipeline for sparsification. Our interface also makes it easy to sparsify existing dense models (e.g., fine-tuning for sparsity) and to add support for additional sparse tensor layouts, operators, or sparsifiers. With STen, we aim to make sparsity easily accessible to ML users and practitioners. As next steps, we plan to extend our implementation to fully support sparse training as well as GPUs and other accelerators.

Acknowledgements

This project received funding from the European Research Council  under the European Union’s Horizon 2020 programme (Project PSAP, No. 101002047); and received EuroHPC-JU funding under grant EU-Pilot, No. 101034126, with support from the Horizon 2020 Programme.

References

- [1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.
- [2] Aart JC Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad. Compiler support for sparse tensor computations in MLIR. *arXiv preprint arXiv:2202.04305*, 2022.
- [3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.
- [4] Erich Elsen, Marat Dukhan, Trevor Gale, and Karen Simonyan. Fast sparse ConvNets. In *Conference on computer vision and pattern recognition (CVPR)*, 2020.
- [5] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations (ICLR)*, 2019.
- [6] Trevor Gale, Erich Elsen, and Sara Hooker. The state of sparsity in deep neural networks. *arXiv preprint arXiv:1902.09574*, 2019.
- [7] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. Sparse GPU kernels for deep learning. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.
- [8] Scott Gray, Alec Radford, and Diederik P Kingma. GPU kernels for block-sparse weights, 2017.
- [9] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *Journal of Machine Learning Research*, 22(241), 2021.
- [10] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.
- [11] Eldar Kurtic, Daniel Campos, Tuan Nguyen, Elias Frantar, Mark Kurtz, Benjamin Fineran, Michael Goin, and Dan Alistarh. The optimal BERT surgeon: Scalable and accurate second-order pruning for large language models. *arXiv preprint arXiv:2203.07259*, 2022.
- [12] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. In *International Conference on Learning Representations (ICLR)*, 2017.
- [13] Christos Louizos, Karen Ullrich, and Max Welling. Bayesian compression for deep learning. In *Advances in neural information processing systems (NeurIPS)*, volume 30, 2017.
- [14] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *International Conference on Learning Representations (ICML)*, 2010.
- [15] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on GPU clusters using Megatron-LM. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2021.
- [16] NVIDIA. NVIDIA A100 tensor core GPU architecture, 2020.
- [17] ONNX. ONNX: Open neural network exchange. <https://onnx.ai/>, 2022.
- [18] OpenAI. AI and compute. <https://openai.com/blog/ai-and-compute/>, 2018.
- [19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems (NeurIPS)*, 2019.

- [20] Jack W Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, et al. Scaling language models: Methods, analysis & insights from training gopher. *arXiv preprint arXiv:2112.11446*, 2021.
- [21] Victor Sanh, Thomas Wolf, and Alexander Rush. Movement pruning: Adaptive sparsity by fine-tuning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.
- [22] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1), 2014.
- [23] Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi. AUGEM: Automatically generate high performance dense linear algebra kernels on x86 CPUs. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.
- [24] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 2020.
- [25] Zhang Xianyi, Wang Qian, and Zhang Yunquan. Model-driven level 3 BLAS performance optimization on Loongson 3A processor. In *International conference on parallel and distributed systems*, 2012.
- [26] Ofir Zafrir, Ariel Larey, Guy Boudoukh, Haihao Shen, and Moshe Wasserblat. Prune once for all: Sparse pre-trained language models. In *Efficient Natural Language and Speech Processing Workshop @ NeurIPS*, 2021.
- [27] AoJun Zhou, Yukun Ma, Junnan Zhu, Jianbo Liu, Zhijie Zhang, Kun Yuan, Wenxiu Sun, and Hongsheng Li. Learning N:M fine-grained structured sparse neural networks from scratch. In *International Conference on Learning Representations (ICLR)*, 2021.
- [28] Michael Zhu and Suyog Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878*, 2017.

A Extended Case Studies

Here we present full details of two case studies using the STen interface. In the first, we sparsify an existing BERT model. In the second, we demonstrate constructing a sparse MLP from scratch. For full details, including notebooks and additional examples, see <https://github.com/spcl/sten>.

A.1 Case study: BERT

modify_existing.ipynb

Here we demonstrate the workflow of adding sparsity into existing models. As an example we take a single encoder layer of BERT [3].

```
import torch
input_shape = (8, 128, 768) # batch, sequence, features
model = torch.hub.load('huggingface/pytorch-transformers',
                      'model', 'bert-base-uncased').encoder.layer[0]
input = torch.rand(input_shape)
output = model(input)
print(output[0].shape) # -> torch.Size([8, 128, 768])
```

We target all linear layers in this model, including feedforward and attention projection layers. A linear layer computes $y = xA^T + b$ and is defined in `torch.nn.Linear` module. In particular, we are going to sparsify tensors A by magnitude pruning of 90% of their values and storing them in the CSR format. In the following snippet we collect the six weight tensors from linear layers, and assign sparsifiers to them.

```
import sten
weights_to_sparsify = []
sb = sten.SparsityBuilder(model)
for module_name, module in model.named_modules():
    if isinstance(module, torch.nn.modules.linear.Linear):
        weight = module_name + ".weight"
        weights_to_sparsify.append(weight)
        sb.set_weight(
            name=weight,
            initial_sparsifier=sten.ScalarFractionSparsifier(0.9),
            inline_sparsifier=sten.KeepAll(),
            tmp_format=torch.Tensor,
            external_sparsifier=sten.KeepAll(),
            out_format=sten.CsrTensor,
        )
print(weights_to_sparsify)
```

This yields the fully qualified names assigned by PyTorch to each of these tensors.

```
['attention.self.query.weight', 'attention.self.key.weight',
 'attention.self.value.weight', 'attention.output.dense.weight',
 'intermediate.dense.weight', 'output.dense.weight']
```

Next, we repeat the same process for intermediate tensors. In this example, we target only the output of the GELU activation. However, it is challenging to refer to this intermediate tensor, as we treat the module as a black box that we do not modify, and internal operators may have varying or no name, depending on the implementation. Examining the layer modules (`print(model)`) shows the model structure:

```
BertLayer(
  ...
  (intermediate): BertIntermediate(
    (dense): Linear(in_features=768, out_features=3072, bias=True)
```

```

    (intermediate_act_fn): GELUActivation()
  )
  ...
)

```

From this we see that the `model.intermediate` submodule contains the GELU activation, but we still do not know the name of the output intermediate tensor. We use the `torch.fx` tracer to assign deterministic names to the intermediate tensors:

```
torch.fx.symbolic_trace(model.intermediate).graph.print_tabular()
```

The result of running this command shows that the output of `<built-in function gelu>` (accessible as `torch.nn.functional.gelu`) is assigned to the tensor with the name `gelu` inside the `model.intermediate` module:

opcode	name	target	args	kwargs
placeholder	hidden_states	hidden_states	()	{}
call_module	dense	dense	(hidden_states,)	{}
call_function	gelu	<built-in function gelu>	(dense,)	{}
output	output	output	(gelu,)	{}

We now assign a random fraction sparsifier with 90% zeroing probability to the GELU output intermediate tensor. The sparsifier stores the tensor in COO format.

```

sb.set_interm(
    name="intermediate.gelu",
    inline_sparsifier=sten.RandomFractionSparsifier(0.9),
    tmp_format=sten.CooTensor,
    external_sparsifier=sten.KeepAll(),
    out_format=sten.CooTensor,
)

```

Finally, we create a new sparse model from the original dense model and run it with the same arguments as before:

```

sparse_model = sb.get_sparse_model()
output = sparse_model(input)
print(output[0].shape) # -> torch.Size([8, 128, 768])

```

A.2 Case study: MLP

[build_from_scratch.ipynb](#)

In this example, we show how to build sparse model from scratch using a simple MLP. As reference we use the following implementation of a dense MLP:

```

import torch

class MLP(torch.nn.Module):
    def __init__(self, channel_sizes):
        super().__init__()
        self.layers = torch.nn.Sequential()
        in_out_pairs = list(zip(channel_sizes[:-1], channel_sizes[1:]))
        for idx, (in_channels, out_channels) in enumerate(in_out_pairs):
            if idx != 0:
                self.layers.append(torch.nn.ReLU())
            self.layers.append(torch.nn.Linear(in_channels, out_channels))

```

```

def forward(self, input):
    return self.layers(input)

model = MLP([50, 40, 30, 20, 30, 10])
output = model(torch.randn(15, 50))
print(output.shape)

```

We are going to replace `torch.nn.Linear` with our custom `SparseLinear` module, which will call our sparse implementation of `torch.nn.functional.linear`.

```

import sten

class SparseLinear(torch.nn.Module):
    def __init__(self, input_features, output_features, weight_sparsity):
        super().__init__()
        self.weight_sparsity = weight_sparsity
        dense_weight = sten.random_mask_sparsify(
            torch.randn(output_features, input_features),
            frac=weight_sparsity
        )
        self.weight = sten.SparseParameterWrapper(
            sten.CscTensor.from_dense(dense_weight)
        )
        self.weight.grad_fmt = (
            sten.KeepAll(),
            torch.Tensor,
            sten.RandomFractionSparsifier(self.weight_sparsity),
            sten.CscTensor,
        )
        self.bias = torch.nn.Parameter(torch.rand(output_features))
        self.bias.grad_fmt = (
            sten.KeepAll(),
            torch.Tensor,
            sten.KeepAll(),
            torch.Tensor,
        )

    def forward(self, input):
        sparse_op = sten.sparsified_op(
            orig_op=torch.nn.functional.linear,
            out_fmt=tuple(
                [(sten.KeepAll(), torch.Tensor,
                  sten.KeepAll(), torch.Tensor)]
            ),
            grad_out_fmt=tuple(
                [(sten.KeepAll(), torch.Tensor,
                  sten.KeepAll(), torch.Tensor)]
            ),
        )
        return sparse_op(input, self.weight, self.bias)

```

The important aspect is the use of `SparseParameterWrapper` to hold the data of sparse tensors. The code above shows the sparsity configuration of weight and intermediate tensors gradients that will appear in the backward pass, although they are dense in this example. The remaining piece is the implementation of `SparseMLP`:

```

class SparseMLP(torch.nn.Module):
    def __init__(self, channel_sizes, weight_sparsity):
        super().__init__()
        self.layers = torch.nn.Sequential()
        in_out_pairs = list(zip(channel_sizes[:-1], channel_sizes[1:]))
        for idx, (in_channels, out_channels) in enumerate(in_out_pairs):
            if idx != 0:
                self.layers.append(torch.nn.ReLU())
            self.layers.append(SparseLinear(
                in_channels, out_channels, weight_sparsity))

    def forward(self, input):
        return self.layers(input)

```

Finally, after the replacement of `torch.nn.Linear` with the `SparseLinear` in the `MLP` implementation, we call it and observe the expected output.

```

model = SparseMLP([50, 40, 30, 20, 30, 10], 0.8)
output = model(torch.randn(15, 50))
print(output.shape)

```

A.3 Case study: Customization

[custom_implementations.ipynb](#)

This example demonstrates the API to register custom operator implementations for specific input and output tensor formats. This example demonstrates customization API to define new sparse tensor formats and sparsifier. It shows how to register custom operator and sparsifier implementations for them.

```

import torch
import sten
import scipy

```

Start from the dense implementation of $d = (a + b)c$.

```

a = torch.randn(10, 20, requires_grad=True)
b = torch.randn(10, 20, requires_grad=True)
c = torch.randn(20, 30, requires_grad=True)
grad_d = torch.randn(10, 30)

d = torch.mm(torch.add(a, b), c)
d.backward(grad_d)

```

First we define a custom random fraction sparsifier functioning the same as `sten.RandomFractionSparsifier`. The sparsifier implementation is not defined here since it is characterized not only by the sparsifier itself but also by the input and output tensor formats. The sparsifier class only needs to declare its configurable parameters.

```

class MyRandomFractionSparsifier:
    def __init__(self, fraction):
        self.fraction = fraction

```

Then declare a tensor in CSC format that will utilize scipy CSC implementation under the hood.

```

class MyCscTensor:
    def __init__(self, data):
        self.data = data

```

```

@staticmethod
def from_dense(tensor):
    return MyCscTensor(scipy.sparse.csc_matrix(tensor))

def to_dense(self):
    return torch.from_numpy(self.data.todense())

```

Then we make the result of addition $a + b$ sparse. To achieve this, we need to replace the addition operator with its sparse counterpart. For simplicity, we do not use an inline sparsifier, which is why the operator outputs a dense `torch.Tensor` after applying the `KeepAll` sparsifier. We use an external random fraction sparsifier with 0.5 dropout probability and output the tensor in the newly defined CSC format. The same specification is assigned to the gradient format, but nothing prevents us from applying a different sparsifier and using a different format for the gradient.

```

sparse_add = sten.sparsified_op(
    orig_op=torch.add,
    out_fmt=(
        (sten.KeepAll(), torch.Tensor,
         MyRandomFractionSparsifier(0.5), MyCscTensor),
    ),
    grad_out_fmt=(
        (sten.KeepAll(), torch.Tensor,
         MyRandomFractionSparsifier(0.5), MyCscTensor),
    ),
)

```

Then we try to use the operator.

```
d = torch.mm(sparse_add(a, b), c)
```

Output:

```

WARNING:root:Sparse operator implementation is not registered (fwd). op:
↳ <built-in method add of type object at 0x7f033a216ea0> inp: (<class
↳ 'torch.Tensor'>, <class 'torch.Tensor'>, None, None) out: ((<class
↳ 'sten.KeepAll'>, <class 'torch.Tensor'>),).. Fallback to dense
↳ implementation.
WARNING:root:Sparsifier implementation is not registered. sparsifier:
↳ <class '__main__.MyRandomFractionSparsifier'> inp: <class
↳ 'torch.Tensor'> out: <class '__main__.MyCscTensor'>. Fallback to dense
↳ keep all implementation.
WARNING:root:Sparse operator implementation is not registered (fwd). op:
↳ <built-in method mm of type object at 0x7f033a216ea0> inp: (<class
↳ '__main__.MyCscTensor'>, <class 'torch.Tensor'>) out: ((<class
↳ 'sten.KeepAll'>, <class 'torch.Tensor'>),).. Fallback to dense
↳ implementation.

```

The first error message indicates the operator implementation which is required is not registered. Here we register it and try calling the method again.

```

@sten.register_fwd_op_impl(
    operator=torch.add,
    inp=(torch.Tensor, torch.Tensor, None, None),
    out=tuple([(sten.KeepAll, torch.Tensor)]),
)
def sparse_add_fwd_impl(ctx, inputs, output_sparsifiers):
    input, other, alpha, out = inputs
    return torch.add(input, other, alpha=alpha, out=out)
d = torch.mm(sparse_add(a, b), c)

```

Output:

```
WARNING:root:Sparsifier implementation is not registered.
sparsifier: <class '__main__.MyRandomFractionSparsifier'>
inp: <class 'torch.Tensor'> out: <class '__main__.MyCscTensor'>.
Fallback to dense keep all implementation.
WARNING:root:Sparse operator implementation is not registered (fwd).
op: <built-in method mm of type object at 0x7f033a216ea0>
inp: (<class '__main__.MyCscTensor'>, <class 'torch.Tensor'>)
out: ((<class 'sten.KeepAll'>, <class 'torch.Tensor'>),).
Fallback to dense implementation.
```

Here we see that sparsifier implementation is not registered. Let's provide it.

```
@sten.register_sparsifier_implementation(
    sparsifier=MyRandomFractionSparsifier, inp=torch.Tensor, out=MyCscTensor
)
def scalar_fraction_sparsifier_dense_coo(sparsifier, tensor):
    return sten.SparseTensorWrapper(
        MyCscTensor.from_dense(
            sten.random_mask_sparsify(tensor, frac=sparsifier.fraction)
        )
    )
d = torch.mm(sparse_add(a, b), c)
```

Output:

```
WARNING:root:Sparse operator implementation is not registered (fwd). op:
→ <built-in method mm of type object at 0x7f033a216ea0> inp: (<class
→ '__main__.MyCscTensor'>, <class 'torch.Tensor'>) out: ((<class
→ 'sten.KeepAll'>, <class 'torch.Tensor'>),).. Fallback to dense
→ implementation.
```

Since $a + b$ is sparse now and it is used as an input of `torch.mm`, we need to provide sparse operator implementation for it as well.

```
@sten.register_fwd_op_impl(
    operator=torch.mm,
    inp=(MyCscTensor, torch.Tensor),
    out=tuple([(sten.KeepAll, torch.Tensor)]),
)
def torch_mm_fwd_impl(ctx, inputs, output_sparsifiers):
    input1, input2 = inputs
    ctx.save_for_backward(input1, input2)
    output = torch.from_numpy(input1.wrapped_tensor.data @ input2.numpy())
    return output
d = torch.mm(sparse_add(a, b), c)
```

As expected, it works. The next step is to call the backward pass and see what remains to be implemented there.

```
d = torch.mm(sparse_add(a, b), c)
try:
    d.backward(grad_d)
except sten.DispatchError as e:
    print(str(e))
```

Output:

```

Sparse operator implementation is not registered (bwd). op: <built-in
→ method mm of type object at 0x7f033a216ea0> grad_out: (<class
→ 'torch.Tensor'>,) grad_inp: ((<class 'sten.KeepAll'>, <class
→ 'torch.Tensor'>), (<class 'sten.KeepAll'>, <class 'torch.Tensor'>))
→ inp: (<class '__main__.MyCscTensor'>, <class 'torch.Tensor'>).

```

Registering the backward implementation for `torch.mm`.

```

@sten.register_bwd_op_impl(
    operator=torch.mm,
    grad_out=(torch.Tensor,),
    grad_inp=(
        (sten.KeepAll, torch.Tensor),
        (sten.KeepAll, torch.Tensor),
    ),
    inp=(MyCscTensor, torch.Tensor),
)
def torch_mm_bwd_impl(ctx, grad_outputs, input_sparsifiers):
    input1, input2 = ctx.saved_tensors
    [grad_output] = grad_outputs
    grad_input1 = torch.mm(grad_output, input2.T)
    grad_input2 = torch.from_numpy(
        input1.wrapped_tensor.data.transpose() @ grad_output)
    return grad_input1, grad_input2

d = torch.mm(sparse_add(a, b), c)
try:
    d.backward(grad_d)
except sten.DispatchError as e:
    print(str(e))

```

Output:

```

Sparse operator implementation is not registered (bwd). op: <built-in
→ method add of type object at 0x7f033a216ea0> grad_out: (<class
→ '__main__.MyCscTensor'>,) grad_inp: ((<class 'sten.KeepAll'>, <class
→ 'torch.Tensor'>), (<class 'sten.KeepAll'>, <class 'torch.Tensor'>),
→ None, None) inp: (<class 'torch.Tensor'>, <class 'torch.Tensor'>, None,
→ None).

```

Backward implementation for `torch.add`:

```

@sten.register_bwd_op_impl(
    operator=torch.add,
    grad_out=(MyCscTensor,),
    grad_inp=(
        (sten.KeepAll, torch.Tensor),
        (sten.KeepAll, torch.Tensor),
        None,
        None,
    ),
    inp=(torch.Tensor, torch.Tensor, None, None),
)
def torch_add_bwd_impl(ctx, grad_outputs, input_sparsifiers):
    [grad_output] = grad_outputs
    dense_output = grad_output.wrapped_tensor.to_dense()
    return dense_output, dense_output, None, None

```

```
d = torch.mm(sparse_add(a, b), c)
d.backward(grad_d)
```

Now backward pass is also fully functional.