STREAMLINING TENSOR AND NETWORK PRUNING IN PYTORCH

Anonymous authors
Paper under double-blind review

ABSTRACT

In order to contrast the explosion in size of state-of-the-art machine learning models that can be attributed to the empirical advantages of over-parametrization, and due to the necessity of deploying fast, sustainable, and private on-device models on resource-constrained devices, the community has focused on techniques such as pruning, quantization, and distillation as central strategies for model compression. Towards the goal of facilitating the adoption of a common interface for neural network pruning in PyTorch, this contribution describes the recent addition of the PyTorch `torch.nn.utils.prune` module, which provides shared, open-source pruning functionalities to lower the technical implementation barrier to reducing model size and capacity before, during, and/or after training. We present the module's user interface, elucidate implementation details, illustrate example usage, and suggest ways to extend the contributed functionalities to new pruning methods.

1 INTRODUCTION

State-of-the-art deep learning techniques rely on over-parametrized models that are hard to deploy. On the contrary, biological neural networks are known to use efficient sparse connectivity (Sporns, 2007; Ramón y Cajal, 1909). Identifying optimal techniques to compress models by reducing the number of parameters is important in order to reduce memory, battery, and hardware consumption without sacrificing accuracy, to deploy lightweight models on device in mobile, IoT, and AR/VR systems, with an eye towards download bandwidth, data consumption, and heat dissipation, and to guarantee privacy with private on-device computation (Nazemi et al., 2018; Yang et al., 2017; Iandola et al., 2016; Han et al., 2015; Reagen et al., 2016; Kim et al., 2015; Verhelst & Moons, 2017). Real-time applications that require reduced latency, meteorological models that, similar to personalization models, require frequent retraining to capture the latest trends, as well as applications with targeted deployment for custom ASICs, may also benefit from the sparsification of models for train and inference speed concerns. Furthermore, the growth in model size has contributed to making reproducing and building upon state-of-the-art techniques only accessible to few, with severe inequalities at a geographical and socioeconomic level. Environmental concerns around the cost and carbon footprint of training large-scale models have been documented in Strubell et al. (2019); Schwartz et al. (2019); Henderson et al. (2020).

Pruning provides ways to remove unnecessary structure in neural networks, thus beginning to address some of the concerns above. Different ways of identifying superfluous portions of a model result in different pruning techniques. These may vary along several axes, including: the nature of the entities to prune (connections, nodes, channels, layers, etc.), the choice of proxy for importance of each entity (weight, activation, gradient, etc.), when to compute the chosen quantity, the group of entities to pool for comparison (all units in the same layer, the whole network, etc.), when to prune (during, before, or after training), whether pruned entities are forever pruned or can be reinstated, whether to apply hard (binary masks) or soft pruning, iterative or one-shot pruning, and what to do with the network once it is pruned (finetuning, reinitializing, rewinding, etc.).

With the addition of the `torch.nn.utils.prune` module[1], PyTorch (Paszke et al., 2019) users may now scan over various choices of pruning techniques as easily as any other choice of hyperparameter and building block in their machine learning pipeline. At the same time, this module

---

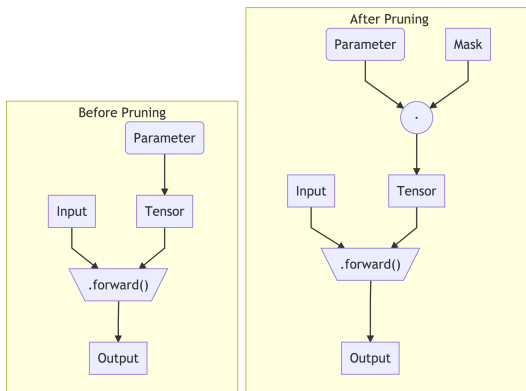[1] Available at https://github.com/pytorch/pytorch/blob/master/torch/nn/utils/prune.py

Figure 1: Reparametrization of a tensor in terms of its unpruned version and the computed mask, and its usage in the forward pass.

aims at empowering researchers to contribute new pruning techniques and express them through a common language.

The goal of this contribution is knowledge dissemination, among the relevant community, of available tools that can simplify and power both research and deployment in resource-constrained scenarios.

2   IMPLEMENTATION

`BasePruningMethod` is an abstract base class that provides a skeleton for all pruning techniques and implements shared functionalities. It enables the creation of new pruning techniques by requiring the overriding of methods such as `compute_mask`. All pruning methods in Sec. 2.1 are derived classes that inherit from it.

The core logic for the application of pruning to a tensor within a module is contained in the the class method `apply`. Specifically, pruning acts by removing the specified parameter from the parameters list and replacing it with a new parameter whose name equals the original one with the string `"_orig"` appended to it. This stores the unpruned version of the tensor. The pruning mask generated by the pruning technique is saved as a module buffer whose name equals the original parameter's name with the string `"_mask"` appended to it. Once the reparametrization is in place, an attribute with the original tensor's name, needed by the `forward` method, is created as a multiplication of the original tensor and the mask by the `apply_mask` method. Finally, the pruning method is attached to the module via a `forward_pre_hook` to ensure that the multiplication op (computed on the fly each time upon calling an instance of the pruning class via `__call__`) is registered into the forward and backward graphs whenever the module is used to compute an output given an input (Fig. 1). The function `torch.nn.utils.prune.is_pruned` returns information about the presence of any pruning hook associated with a module. All relevant tensors, including the mask buffers and the original parameters used to compute the pruned tensors are stored in the model's `state_dict` and can therefore be easily serialized and saved.

In case the reparametrization and hook creation fail, the related exception is raised and the module is rolled back to its state prior to the failed pruning attempt, without compromising its usability.

To remove the reparametrization depicted in Fig. 1 and make the pruning permanent, the user can call the function `torch.nn.utils.prune.remove`, which removes buffers, hooks, and additional attributes, and assigns the pruned tensor to the parameter with the original parameter name. Note that pruning itself is not undone or reversed by this operation.

A `PruningContainer` is used to store the history of pruning calls executed on a module in order to enable iterative pruning, *i.e.* the sequential application of pruning techniques. Each parameter in a module that is pruned more than once has an associated `PruningContainer`; that container has an instance attribute called `_tensor_name` that identifies which parameter in the module it

relates to. Only pruning methods that act on the same tensor can be added to the container. The tuple attribute `_pruning_methods` stores the instances of pruning techniques in the order they are applied.

Each pruning method has a `PRUNING_TYPE` that dictates how to combine iterative pruning masks. At the moment, this supports the following types: unstructured, structured, and global. An unstructured pruning technique disregards individual entries in a tensor that have already been pruned; a structured pruning technique disregards a row or column only if all its entries have already been pruned; a global pruning method, such a `torch.nn.utils.prune.CustomFromMask` in Sec. 2.1, applies the pruning technique to all entries, regardless of whether they have been pruned before. These code paths are defined in the `_combine_masks` inner utility function within the `PruningContainer`'s `compute_mask` method. For most other intents and purposes, a container works like any other derived class of a `BasePruningMethod`.

The `torch.nn.utils.prune` module also provides a simple and clean functional interface that allows users to interact with pruning techniques through intuitive function calls on modules' parameters identified by name. See Sec. 3 for example usage.

2.1 AVAILABLE PRUNING METHODS

The following child classes inherit from the `BasePruningMethod`:

- `torch.nn.utils.prune.Identity`: utility pruning method that does not prune any units but generates the pruning parametrization with a mask of ones;
- `torch.nn.utils.prune.RandomUnstructured`: prune (currently unpruned) entries in a tensor at random;
- `torch.nn.utils.prune.L1Unstructured`: prune (currently unpruned) entries in a tensor by zeroing out the ones with the lowest absolute magnitude;
- `torch.nn.utils.prune.RandomStructured`: prune entire (currently unpruned) rows or columns in a tensor at random;
- `torch.nn.utils.prune.LnStructured`: prune entire (currently unpruned) rows or columns in a tensor based on their L_n-norm (supported values of n correspond to supported values for argument p in `torch.norm()`);
- `torch.nn.utils.prune.CustomFromMask`: prune a tensor using a user-provided mask.

Their functional equivalents are:

- `torch.nn.utils.prune.identity`
- `torch.nn.utils.prune.random_unstructured`
- `torch.nn.utils.prune.l1_unstructured`
- `torch.nn.utils.prune.random_structured`
- `torch.nn.utils.prune.ln_structured`
- `torch.nn.utils.prune.custom_from_mask`

Global pruning, in which entries are compared across multiple tensors, is enabled through `torch.nn.utils.prune.global_unstructured` (see Sec. 3 for example usage).

2.2 EXTENDING THE MODULE

The `torch.nn.utils.prune` module can be extended to implement custom pruning functions. This requires sub-classing the `BasePruningMethod` base class, and implementing the `__init__` and `compute_mask` methods, *i.e.*, respectively, the constructor and the instructions to compute the mask for the given tensor according to the logic of the pruning technique. The nature of the pruning technique is specified through the assignment of a `PRUNING_TYPE`. If none of the currently supported types fits the new pruning technique, the user will also have to add support for a new

`PRUNING_TYPE` in the way `PruningContainer` handles the iterative application of pruning masks.

For example, Listing 1 demonstrates how to implement a pruning technique that prunes every other currently unpruned entry in a tensor, and how to provide a convenient functional interface for the method in just a few lines of code.

```python
import torch.nn.utils.prune as prune

class FooBarPruningMethod(prune.BasePruningMethod):
    """Prune every other entry in a tensor
    """
    PRUNING_TYPE = 'unstructured'

    def compute_mask(self, t, default_mask):
        mask = default_mask.clone()
        mask.view(-1)[::2] = 0
        return mask

def foobar_unstructured(module, name):
    """Prunes tensor corresponding to parameter called `name` in `module`
    by removing every other entry in the tensors.
    Modifies module in place (and also return the modified module)
    by:
    1) adding a named buffer called `name+'_mask'` corresponding to the
    binary mask applied to the parameter `name` by the pruning method.
    The parameter `name` is replaced by its pruned version, while the
    original (unpruned) parameter is stored in a new parameter named
    `name+'_orig'`.
    Args:
        module (nn.Module): module containing the tensor to prune
        name (string): parameter name within `module` on which pruning
                will act.
    Returns:
        module (nn.Module): modified (i.e. pruned) version of the input
            module

    Examples:
        >>> m = nn.Linear(3, 4)
        >>> foobar_unstructured(m, name='bias')
    """
    FooBarPruningMethod.apply(module, name)
    return module
```

Listing 1: Simple extension to the `torch.nn.utils.prune` module that implements a custom pruning function to prune every other unpruned entry in a parameter.

3 EXAMPLE USAGE

Pruning parameters in a model is as simple as invoking the desired pruning function on each parameter (identified by name) within a given neural network module. In this example, the first convolutional layer in a VGG-11 architecture (Simonyan & Zisserman, 2014) is first pruned by removing 3 individual entries using unstructured magnitude-based pruning, then pruned again by removing the bottom 50% of remaining channels along the 0th axis by L_2-norm.

```python
import torch
from torch import nn
import torch.nn.functional as F
from torchvision.models import vgg11

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = vgg11().to(device=device)

# Prune the first convolutional layer
prune.l1_unstructured(model.features[0], name="weight", amount=3)

# Iteratively prune by simply calling another pruning function on the same parameter.
prune.ln_structured(model.features[0], name="weight", amount=0.5, n=2, dim=0)
```

Listing 2: Simple iterative pruning of a single parameter in a network.

This can be easily extended to apply pruning to all layers in a network. For instance, Listing 3 shows how to prune all 2D convolutional and linear layers in a network, with different pruning fractions that depend on the layer type.

```python
new_model = vgg11().to(device=device)
for name, module in new_model.named_modules():
    # Prune 20% of connections in all 2D-conv layers
    if isinstance(module, torch.nn.Conv2d):
        prune.l1_unstructured(module, name='weight', amount=0.2)
```

```
6    # Prune 40% of connections in all linear layers
7    elif isinstance(module, torch.nn.Linear):
8        prune.l1_unstructured(module, name='weight', amount=0.4)
```

Listing 3: Automated pruning of all weights in a network belonging to specific layer types.

In the examples above, each candidate entity for pruning is compared in magnitude to other candidate entities within a single layer. Listing 4, instead, provides an example of how to enable the pooling together of all entities (*i.e.* single connections, entire units, or channels) across a network for a global magnitude comparison.

```
1  class LeNet(nn.Module):
2      def __init__(self):
3          super(LeNet, self).__init__()
4          # 1 input image channel, 6 output channels, 3x3 square conv kernel
5          self.conv1 = nn.Conv2d(1, 6, 3)
6          self.conv2 = nn.Conv2d(6, 16, 3)
7          self.fc1 = nn.Linear(16 * 5 * 5, 120)   # 5x5 image dimension
8          self.fc2 = nn.Linear(120, 84)
9          self.fc3 = nn.Linear(84, 10)
10
11     def forward(self, x):
12         x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
13         x = F.max_pool2d(F.relu(self.conv2(x)), 2)
14         x = x.view(-1, int(x.nelement() / x.shape[0]))
15         x = F.relu(self.fc1(x))
16         x = F.relu(self.fc2(x))
17         x = self.fc3(x)
18         return x
19
20 model = LeNet()
21
22 parameters_to_prune = (
23     (model.conv1, 'weight'),
24     (model.conv2, 'weight'),
25     (model.fc1, 'weight'),
26     (model.fc2, 'weight'),
27     (model.fc3, 'weight'),
28 )
29
30 prune.global_unstructured(
31     parameters_to_prune,
32     pruning_method=prune.L1Unstructured,
33     amount=0.2,
34 )
```

Listing 4: Example of how to prune the bottom 20% of connections by absolute magnitude across an entire LeNet (LeCun et al., 1998) architecture.

The module also supports the pruning of individual tensors by calling the `prune` method of any pruning class, as shown in Listing 5.

```
1 t = torch.rand(2, 5)
2 p = prune.RandomUnstructured(amount=0.7)
3 pruned_tensor = p.prune(t)
```

Listing 5: Code to prune 70% of entries at random in a tensor not associated with any `torch.nn.Module`.

REFERENCES

S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, 2015.

P. Henderson et al. Towards the systematic reporting of the energy and carbon footprints of machine learning. 2020.

F. N. Iandola et al. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and ¡0.5mb model size, 2016.

Y.-D. Kim et al. Compression of deep convolutional neural networks for fast and low power mobile applications, 2015.

Y. LeCun et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

M. Nazemi, G. Pasandi, and M. Pedram. Nullanet: Training deep neural networks for reduced-memory-access inference. *CoRR*, abs/1807.08716, 2018. URL `http://arxiv.org/abs/1807.08716`.

A. Paszke et al. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019. URL `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

S. Ramón y Cajal. *Histologie du système nerveux de l'homme des vertébrés.*, volume v. 1. Paris :Maloine,, 1909. URL `https://www.biodiversitylibrary.org/item/103261`.

B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G. Wei, and D. Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 267–278, June 2016. doi: 10.1109/ISCA.2016.32.

R. Schwartz et al. Green ai, 2019.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2014.

O. Sporns. Brain connectivity. *Scholarpedia*, 2(10):4695, 2007. doi: 10.4249/scholarpedia.4695. revision #91084.

E. Strubell, A. Ganesh, and A. McCallum. Energy and policy considerations for deep learning in nlp, 2019.

M. Verhelst and B. Moons. Embedded deep neural network processing: Algorithmic and processor techniques bring deep learning to iot and edge devices. *IEEE Solid-State Circuits Magazine*, 9 (4):55–65, Fall 2017. ISSN 1943-0590. doi: 10.1109/MSSC.2017.2745818.

T.-J. Yang, Y.-H. Chen, and V. Sze. Designing energy-efficient convolutional neural networks using energy-aware pruning. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.