Table of Contents

Colab Notebook GitHub

Pruning Tutorial

Author: Michela Paganini

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. Identifying optimal techniques to compress models by reducing the number of parameters in them is important in order to reduce memory, battery, and hardware consumption without sacrificing accuracy. This in turn allows you to deploy lightweight models on device, and guarantee privacy with private on-device computation. On the research front, pruning is used to investigate the differences in learning dynamics between over-parametrized and under-parametrized networks, to study the role of lucky sparse subnetworks and initializations ("lottery tickets") as a destructive neural architecture search technique, and more.

In this tutorial, you will learn how to use `torch.nn.utils.prune` to sparsify your neural networks, and how to extend it to implement your own custom pruning technique.

Requirements

`"torch>=1.4.0a0+8e8a5e0"`

```
import torch
from torch import nn
import torch.nn.utils.prune as prune
import torch.nn.functional as F
```

Create a model

In this tutorial, we use the LeNet architecture from LeCun et al., 1998.

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        # 1 input image channel, 6 output channels, 5x5 square conv kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)  # 5x5 image dimension
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, int(x.nelement() / x.shape[0]))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

model = LeNet().to(device=device)
```

Inspect a Module

Let's inspect the (unpruned) `conv1` layer in our LeNet model. It will contain two parameters `weight` and `bias`, and no buffers, for now.

```
module = model.conv1
print(list(module.named_parameters()))
```

Out:

```
          [-0.0475,  0.1144, -0.1554, -0.1009,  0.0610],
          [ 0.0423, -0.0510,  0.1192,  0.1360, -0.1450],
          [-0.1068,  0.1831, -0.0675, -0.0709, -0.1935]]],


         [[[-0.1145,  0.0500, -0.0264, -0.1452,  0.0047],
          [-0.1366, -0.1697, -0.1101, -0.1750, -0.1273],
          [ 0.1999,  0.0378,  0.0616, -0.1865, -0.1314],
          [-0.0666,  0.0313, -0.1760, -0.0862, -0.1197],
          [ 0.0006, -0.0744, -0.0139, -0.1355, -0.1373]]],


         [[[-0.1167, -0.0685, -0.1579,  0.1677, -0.0397],
          [ 0.1721,  0.0623, -0.1694,  0.1384, -0.0550],
          [-0.0767, -0.1660, -0.1988,  0.0572, -0.0437],
          [ 0.0779, -0.1641,  0.1485, -0.1468, -0.0345],
          [ 0.0418,  0.1033,  0.1615,  0.1822, -0.1586]]]], device='cuda:0',
       requires_grad=True)), ('bias', Parameter containing:
tensor([ 0.0503, -0.0860, -0.0219, -0.1497,  0.1822, -0.1468], device='cuda:0',
       requires_grad=True))]
```

```
print(list(module.named_buffers()))
```

Out:

```
[]
```

Pruning a Module

To prune a module (in this example, the `conv1` layer of our LeNet architecture), first select a pruning technique among those available in `torch.nn.utils.prune` (or implement your own by subclassing `BasePruningMethod`). Then, specify the module and the name of the parameter to prune within that module. Finally, using the adequate keyword arguments required by the selected pruning technique, specify the pruning parameters.

In this example, we will prune at random 30% of the connections in the parameter named `weight` in the `conv1` layer. The module is passed as the first argument to the function; `name` identifies the parameter within that module using its string identifier; and `amount` indicates either the percentage of connections to prune (if it is a float between 0. and 1.), or the absolute number of connections to prune (if it is a non-negative integer).

```
prune.random_unstructured(module, name="weight", amount=0.3)
```

Out:

```
Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
```

Pruning acts by removing `weight` from the parameters and replacing it with a new parameter called `weight_orig` (i.e. appending `"_orig"` to the initial parameter `name`). `weight_orig` stores the unpruned version of the tensor. The `bias` was not pruned, so it will remain intact.

```
print(list(module.named_parameters()))
```

Out:

```
[('bias', Parameter containing:
tensor([ 0.0503, -0.0860, -0.0219, -0.1497,  0.1822, -0.1468], device='cuda:0',
       requires_grad=True)), ('weight_orig', Parameter containing:
tensor([[[[ 0.1529,  0.1660, -0.0469,  0.1837, -0.0438],
          [ 0.0404, -0.0974,  0.1175,  0.1763, -0.1467],
          [ 0.1738,  0.0374,  0.1478,  0.0271,  0.0964],
          [-0.0282,  0.1542,  0.0296, -0.0934,  0.0510],
          [-0.0921, -0.0235, -0.0812,  0.1327, -0.1579]]],


         [[[-0.0922, -0.0565, -0.1203,  0.0189, -0.1975],
          [ 0.1806, -0.1699,  0.1544,  0.0333, -0.0649],
          [ 0.1236,  0.0312,  0.1616,  0.0219, -0.0631],
          [ 0.0537, -0.0542,  0.0842,  0.1786,  0.1156],
          [-0.0874,  0.1155,  0.0358,  0.1016, -0.1219]]],


         [[[-0.1980, -0.0773, -0.1534,  0.1641,  0.0576],
          [ 0.0828,  0.0633, -0.0035,  0.1565, -0.1421],
          [ 0.0126, -0.1365,  0.0617, -0.0689,  0.0613],
```

The pruning mask generated by the pruning technique selected above is saved as a module buffer named `weight_mask` (i.e. appending `"_mask"` to the initial parameter `name`).

```
print(list(module.named_buffers()))
```

Out:

```
[('weight_mask', tensor([[[[1., 1., 1., 1., 1.],
          [1., 0., 1., 1., 1.],
          [1., 0., 0., 1., 1.],
          [1., 0., 1., 1., 1.],
          [1., 0., 0., 1., 1.]]],


        [[[1., 1., 1., 0., 1.],
          [1., 1., 1., 1., 1.],
          [0., 1., 1., 1., 0.],
          [1., 1., 0., 1., 0.],
          [0., 1., 0., 1., 1.]]],


        [[[1., 0., 0., 0., 1.],
          [1., 0., 1., 1., 0.],
          [1., 1., 1., 1., 1.],
          [1., 1., 1., 1., 1.],
          [1., 0., 1., 1., 0.]]],
```

For the forward pass to work without modification, the `weight` attribute needs to exist. The pruning techniques implemented in `torch.nn.utils.prune` compute the pruned version of the weight (by combining the mask with the original parameter) and store them in the attribute `weight`. Note, this is no longer a parameter of the `module`, it is now simply an attribute.

```
print(module.weight)
```

Out:

```
tensor([[[[ 0.1529,  0.1660, -0.0469,  0.1837, -0.0438],
          [ 0.0404, -0.0000,  0.1175,  0.1763, -0.1467],
          [ 0.1738,  0.0000,  0.0000,  0.0271,  0.0964],
          [-0.0282,  0.0000,  0.0296, -0.0934,  0.0510],
          [-0.0921, -0.0000, -0.0000,  0.1327, -0.1579]]],


        [[[-0.0922, -0.0565, -0.1203,  0.0000, -0.1975],
          [ 0.1806, -0.1699,  0.1544,  0.0333, -0.0649],
          [ 0.0000,  0.0312,  0.1616,  0.0219, -0.0000],
          [ 0.0537, -0.0542,  0.0000,  0.1786,  0.0000],
          [-0.0000,  0.1155,  0.0000,  0.1016, -0.1219]]],


        [[[-0.1980, -0.0000, -0.0000,  0.0000,  0.0576],
          [ 0.0828,  0.0000, -0.0035,  0.1565, -0.0000],
          [ 0.0126, -0.1365,  0.0617, -0.0689,  0.0613],
          [-0.0417,  0.1659, -0.1185, -0.1193, -0.1193],
          [ 0.1799,  0.0000,  0.1925, -0.1651, -0.0000]]],
```

Finally, pruning is applied prior to each forward pass using PyTorch's `forward_pre_hooks`. Specifically, when the `module` is pruned, as we have done here, it will acquire a `forward_pre_hook` for each parameter associated with it that gets pruned. In this case, since we have so far only pruned the original parameter named `weight`, only one hook will be present.

```
print(module._forward_pre_hooks)
```

Out:

```
OrderedDict([(3, <torch.nn.utils.prune.RandomUnstructured object at 0x7f58733971f0>)])
```

For completeness, we can now prune the `bias` too, to see how the parameters, buffers, hooks, and attributes of the `module` change. Just for the sake of trying out another pruning technique, here we prune the 3 smallest entries in the bias by L1 norm, as implemented in the `l1_unstructured` pruning function.

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

Out:

```
Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
```

We now expect the named parameters to include both `weight_orig` (from before) and `bias_orig`. The buffers will include `weight_mask` and `bias_mask`. The pruned versions of the two tensors will exist as module attributes, and the module will now have two `forward_pre_hooks`.

```
print(list(module.named_parameters()))
```

Out:

```
[('weight_orig', Parameter containing:
tensor([[[[ 0.1529,  0.1660, -0.0469,  0.1837, -0.0438],
          [ 0.0404, -0.0974,  0.1175,  0.1763, -0.1467],
          [ 0.1738,  0.0374,  0.1478,  0.0271,  0.0964],
          [-0.0282,  0.1542,  0.0296, -0.0934,  0.0510],
          [-0.0921, -0.0235, -0.0812,  0.1327, -0.1579]]],


        [[[-0.0922, -0.0565, -0.1203,  0.0189, -0.1975],
          [ 0.1806, -0.1699,  0.1544,  0.0333, -0.0649],
          [ 0.1236,  0.0312,  0.1616,  0.0219, -0.0631],
          [ 0.0537, -0.0542,  0.0842,  0.1786,  0.1156],
          [-0.0874,  0.1155,  0.0358,  0.1016, -0.1219]]],


        [[[-0.1980, -0.0773, -0.1534,  0.1641,  0.0576],
          [ 0.0828,  0.0633, -0.0035,  0.1565, -0.1421],
          [ 0.0126, -0.1365,  0.0617, -0.0689,  0.0613],
          [-0.0417,  0.1659, -0.1185, -0.1193, -0.1193],
          [ 0.1799,  0.0667,  0.1925, -0.1651, -0.1984]]],
```

```
print(list(module.named_buffers()))
```

Out:

```
[('weight_mask', tensor([[[[1., 1., 1., 1., 1.],
          [1., 0., 1., 1., 1.],
          [1., 0., 0., 1., 1.],
          [1., 0., 1., 1., 1.],
          [1., 0., 0., 1., 1.]]],


        [[[1., 1., 1., 0., 1.],
          [1., 1., 1., 1., 1.],
          [0., 1., 1., 1., 0.],
          [1., 1., 0., 1., 0.],
          [0., 1., 0., 1., 1.]]],


        [[[1., 0., 0., 0., 1.],
          [1., 0., 1., 1., 0.],
          [1., 1., 1., 1., 1.],
          [1., 1., 1., 1., 1.],
          [1., 0., 1., 1., 0.]]],
```

```
print(module.bias)
```

Out:

```
tensor([ 0.0000, -0.0000, -0.0000, -0.1497,  0.1822, -0.1468], device='cuda:0',
       grad_fn=<MulBackward0>)
```

```
print(module._forward_pre_hooks)
```

Out:

```
OrderedDict([(3, <torch.nn.utils.prune.RandomUnstructured object at 0x7f58733971f0>), (4, <torch.nn.utils.prune.L1Unstructured object
at 0x7f5873396830>)])
```

Iterative Pruning

The same parameter in a module can be pruned multiple times, with the effect of the various pruning calls being equal to the combination of the various masks applied in series. The combination of a new mask with the old mask is handled by the `PruningContainer`'s `compute_mask` method.

Say, for example, that we now want to further prune `module.weight`, this time using structured pruning along the 0th axis of the tensor (the 0th axis corresponds to the output channels of the convolutional layer and has dimensionality 6 for `conv1`), based on the channels' L2 norm. This can be achieved using the `ln_structured` function, with `n=2` and `dim=0`.

```python
prune.ln_structured(module, name="weight", amount=0.5, n=2, dim=0)

# As we can verify, this will zero out all the connections corresponding to
# 50% (3 out of 6) of the channels, while preserving the action of the
# previous mask.
print(module.weight)
```

Out:

```
tensor([[[[ 0.0000,  0.0000, -0.0000,  0.0000, -0.0000],
          [ 0.0000, -0.0000,  0.0000,  0.0000, -0.0000],
          [ 0.0000,  0.0000,  0.0000,  0.0000,  0.0000],
          [-0.0000,  0.0000,  0.0000, -0.0000,  0.0000],
          [-0.0000, -0.0000, -0.0000,  0.0000, -0.0000]]],


        [[[-0.0000, -0.0000, -0.0000,  0.0000, -0.0000],
          [ 0.0000, -0.0000,  0.0000,  0.0000, -0.0000],
          [ 0.0000,  0.0000,  0.0000,  0.0000, -0.0000],
          [ 0.0000, -0.0000,  0.0000,  0.0000,  0.0000],
          [-0.0000,  0.0000,  0.0000,  0.0000, -0.0000]]],


        [[[-0.1980, -0.0000, -0.0000,  0.0000,  0.0576],
          [ 0.0828,  0.0000, -0.0035,  0.1565, -0.0000],
          [ 0.0126, -0.1365,  0.0617, -0.0689,  0.0613],
          [-0.0417,  0.1659, -0.1185, -0.1193, -0.1193],
          [ 0.1799,  0.0000,  0.1925, -0.1651, -0.0000]]],
```

The corresponding hook will now be of type `torch.nn.utils.prune.PruningContainer`, and will store the history of pruning applied to the `weight` parameter.

```python
for hook in module._forward_pre_hooks.values():
    if hook._tensor_name == "weight":  # select out the correct hook
        break

print(list(hook))  # pruning history in the container
```

Out:

```
[<torch.nn.utils.prune.RandomUnstructured object at 0x7f58733971f0>, <torch.nn.utils.prune.LnStructured object at 0x7f5873396a70>]
```

Serializing a pruned model

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, if needed.

```python
print(model.state_dict().keys())
```

Out:

```
odict_keys(['conv1.weight_orig', 'conv1.bias_orig', 'conv1.weight_mask', 'conv1.bias_mask', 'conv2.weight', 'conv2.bias', 'fc1.weight',
'fc1.bias', 'fc2.weight', 'fc2.bias', 'fc3.weight', 'fc3.bias'])
```

Remove pruning re-parametrization

To make the pruning permanent, remove the re-parametrization in terms of `weight_orig` and `weight_mask`, and remove the `forward_pre_hook`, we can use the `remove` functionality from `torch.nn.utils.prune`. Note that this doesn't undo the pruning, as if it never happened. It simply makes it permanent, instead, by reassigning the parameter `weight` to the model parameters, in its pruned version.

Prior to removing the re-parametrization:

```
print(list(module.named_parameters()))
```

Out:

```
[('weight_orig', Parameter containing:
tensor([[[[ 0.1529,  0.1660, -0.0469,  0.1837, -0.0438],
          [ 0.0404, -0.0974,  0.1175,  0.1763, -0.1467],
          [ 0.1738,  0.0374,  0.1478,  0.0271,  0.0964],
          [-0.0282,  0.1542,  0.0296, -0.0934,  0.0510],
          [-0.0921, -0.0235, -0.0812,  0.1327, -0.1579]]],


        [[[-0.0922, -0.0565, -0.1203,  0.0189, -0.1975],
          [ 0.1806, -0.1699,  0.1544,  0.0333, -0.0649],
          [ 0.1236,  0.0312,  0.1616,  0.0219, -0.0631],
          [ 0.0537, -0.0542,  0.0842,  0.1786,  0.1156],
          [-0.0874,  0.1155,  0.0358,  0.1016, -0.1219]]],


        [[[-0.1980, -0.0773, -0.1534,  0.1641,  0.0576],
          [ 0.0828,  0.0633, -0.0035,  0.1565, -0.1421],
          [ 0.0126, -0.1365,  0.0617, -0.0689,  0.0613],
          [-0.0417,  0.1659, -0.1185, -0.1193, -0.1193],
          [ 0.1799,  0.0667,  0.1925, -0.1651, -0.1984]]],
```

```
print(list(module.named_buffers()))
```

Out:

```
[('weight_mask', tensor([[[[0., 0., 0., 0., 0.],
          [0., 0., 0., 0., 0.],
          [0., 0., 0., 0., 0.],
          [0., 0., 0., 0., 0.],
          [0., 0., 0., 0., 0.]]],


        [[[0., 0., 0., 0., 0.],
          [0., 0., 0., 0., 0.],
          [0., 0., 0., 0., 0.],
          [0., 0., 0., 0., 0.],
          [0., 0., 0., 0., 0.]]],


        [[[1., 0., 0., 0., 1.],
          [1., 0., 1., 1., 0.],
          [1., 1., 1., 1., 1.],
          [1., 1., 1., 1., 1.],
          [1., 0., 1., 1., 0.]]],
```

```
print(module.weight)
```

Out:

```
tensor([[[[ 0.0000,  0.0000, -0.0000,  0.0000, -0.0000],
          [ 0.0000, -0.0000,  0.0000,  0.0000, -0.0000],
          [ 0.0000,  0.0000,  0.0000,  0.0000,  0.0000],
          [-0.0000,  0.0000,  0.0000, -0.0000,  0.0000],
          [-0.0000, -0.0000, -0.0000,  0.0000, -0.0000]]],


        [[[-0.0000, -0.0000, -0.0000,  0.0000, -0.0000],
          [ 0.0000, -0.0000,  0.0000,  0.0000, -0.0000],
          [ 0.0000,  0.0000,  0.0000,  0.0000, -0.0000],
          [ 0.0000, -0.0000,  0.0000,  0.0000,  0.0000],
          [-0.0000,  0.0000,  0.0000,  0.0000, -0.0000]]],


        [[[-0.1980, -0.0000, -0.0000,  0.0000,  0.0576],
          [ 0.0828,  0.0000, -0.0035,  0.1565, -0.0000],
          [ 0.0126, -0.1365,  0.0617, -0.0689,  0.0613],
          [-0.0417,  0.1659, -0.1185, -0.1193, -0.1193],
          [ 0.1799,  0.0000,  0.1925, -0.1651, -0.0000]]],
```

After removing the re-parametrization:

```
prune.remove(module, 'weight')
print(list(module.named_parameters()))
```

Out:

```
[('bias_orig', Parameter containing:
tensor([ 0.0503, -0.0860, -0.0219, -0.1497,  0.1822, -0.1468], device='cuda:0',
       requires_grad=True)), ('weight', Parameter containing:
tensor([[[[ 0.0000,  0.0000, -0.0000,  0.0000, -0.0000],
          [ 0.0000, -0.0000,  0.0000,  0.0000, -0.0000],
          [ 0.0000,  0.0000,  0.0000,  0.0000,  0.0000],
          [-0.0000,  0.0000,  0.0000, -0.0000,  0.0000],
          [-0.0000, -0.0000, -0.0000,  0.0000, -0.0000]]],


        [[[-0.0000, -0.0000, -0.0000,  0.0000, -0.0000],
          [ 0.0000, -0.0000,  0.0000,  0.0000, -0.0000],
          [ 0.0000,  0.0000,  0.0000,  0.0000, -0.0000],
          [ 0.0000, -0.0000,  0.0000,  0.0000,  0.0000],
          [-0.0000,  0.0000,  0.0000,  0.0000, -0.0000]]],


        [[[-0.1980, -0.0000, -0.0000,  0.0000,  0.0576],
          [ 0.0828,  0.0000, -0.0035,  0.1565, -0.0000],
          [ 0.0126, -0.1365,  0.0617, -0.0689,  0.0613],
```

```
print(list(module.named_buffers()))
```

Out:

```
[('bias_mask', tensor([0., 0., 0., 1., 1., 1.], device='cuda:0'))]
```

Pruning multiple parameters in a model

By specifying the desired pruning technique and parameters, we can easily prune multiple tensors in a network, perhaps according to their type, as we will see in this example.

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

print(dict(new_model.named_buffers()).keys())  # to verify that all masks exist
```

Out:

```
dict_keys(['conv1.weight_mask', 'conv2.weight_mask', 'fc1.weight_mask', 'fc2.weight_mask', 'fc3.weight_mask'])
```

Global pruning

So far, we only looked at what is usually referred to as "local" pruning, i.e. the practice of pruning tensors in a model one by one, by comparing the statistics (weight magnitude, activation, gradient, etc.) of each entry exclusively to the other entries in that tensor. However, a common and perhaps more powerful technique is to prune the model all at once, by removing (for example) the lowest 20% of connections across the whole model, instead of removing the lowest 20% of connections in each layer. This is likely to result in different pruning percentages per layer. Let's see how to do that using `global_unstructured` from `torch.nn.utils.prune`.

```
model = LeNet()

parameters_to_prune = (
    (model.conv1, 'weight'),
    (model.conv2, 'weight'),
    (model.fc1, 'weight'),
    (model.fc2, 'weight'),
    (model.fc3, 'weight'),
)

prune.global_unstructured(
    parameters_to_prune,
    pruning_method=prune.L1Unstructured,
    amount=0.2,
)
```

Now we can check the sparsity induced in every pruned parameter, which will not be equal to 20% in each layer. However, the global sparsity will be (approximately) 20%.

```
print(
    "Sparsity in conv1.weight: {:.2f}%".format(
        100. * float(torch.sum(model.conv1.weight == 0))
        / float(model.conv1.weight.nelement())
    )
)
print(
    "Sparsity in conv2.weight: {:.2f}%".format(
        100. * float(torch.sum(model.conv2.weight == 0))
        / float(model.conv2.weight.nelement())
    )
)
print(
    "Sparsity in fc1.weight: {:.2f}%".format(
        100. * float(torch.sum(model.fc1.weight == 0))
        / float(model.fc1.weight.nelement())
    )
)
print(
    "Sparsity in fc2.weight: {:.2f}%".format(
        100. * float(torch.sum(model.fc2.weight == 0))
        / float(model.fc2.weight.nelement())
    )
)
print(
    "Sparsity in fc3.weight: {:.2f}%".format(
        100. * float(torch.sum(model.fc3.weight == 0))
        / float(model.fc3.weight.nelement())
    )
)
print(
    "Global sparsity: {:.2f}%".format(
        100. * float(
            torch.sum(model.conv1.weight == 0)
            + torch.sum(model.conv2.weight == 0)
            + torch.sum(model.fc1.weight == 0)
            + torch.sum(model.fc2.weight == 0)
            + torch.sum(model.fc3.weight == 0)
        )
        / float(
            model.conv1.weight.nelement()
            + model.conv2.weight.nelement()
            + model.fc1.weight.nelement()
            + model.fc2.weight.nelement()
            + model.fc3.weight.nelement()
        )
    )
)
```

Out:

```
Sparsity in conv1.weight: 4.67%
Sparsity in conv2.weight: 13.92%
Sparsity in fc1.weight: 22.16%
Sparsity in fc2.weight: 12.10%
Sparsity in fc3.weight: 11.31%
Global sparsity: 20.00%
```

Extending `torch.nn.utils.prune` with custom pruning functions

To implement your own pruning function, you can extend the `nn.utils.prune` module by subclassing the `BasePruningMethod` base class, the same way all other pruning methods do. The base class implements the following methods for you: `__call__`, `apply_mask`, `apply`, `prune`, and `remove`. Beyond some special cases, you shouldn't have to reimplement these methods for your new pruning technique. You will, however, have to implement `__init__` (the constructor), and `compute_mask` (the instructions on how to compute the mask for the given tensor according to the logic of your pruning technique). In addition, you will have to specify which type of pruning this technique implements (supported options are `global`, `structured`, and `unstructured`). This is needed to determine how to combine masks in the case in which pruning is applied iteratively. In other words, when pruning a prepruned parameter, the current pruning technique is expected to act on the unpruned portion of the parameter. Specifying the `PRUNING_TYPE` will enable the `PruningContainer` (which handles the iterative application of pruning masks) to correctly identify the slice of the parameter to prune.

Let's assume, for example, that you want to implement a pruning technique that prunes every other entry in a tensor (or – if the tensor has previously been pruned – in the remaining unpruned portion of the tensor). This will be of `PRUNING_TYPE='unstructured'` because it acts on individual connections in a layer and not on entire units/channels (`'structured'`), or across different parameters (`'global'`).

```python
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
```

Now, to apply this to a parameter in an `nn.Module`, you should also provide a simple function that instantiates the method and applies it.

```python
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
```

Let's try it out!

```python
model = LeNet()
foobar_unstructured(model.fc3, name='bias')

print(model.fc3.bias_mask)
```

Out:

```
tensor([0., 1., 0., 1., 0., 1., 0., 1., 0., 1.])
```

Total running time of the script: (0 minutes 0.308 seconds)