In [1]:
```
# For tips on running notebooks in Google Colab, see
# https://pytorch.org/tutorials/beginner/colab
%matplotlib inline
```

剪枝教程

作者：Michela Paganini

最先进的深度学习技术依赖于难以部署的过度参数化模型。相反，已知生物神经网络使用有效的稀疏连接。为了在不牺牲精度的情况下减少内存、电池和硬件消耗，通过减少模型中的参数数量来确定压缩模型的最佳技术是很重要的。这反过来又允许在设备上部署轻量级模型，并通过设备上的私有计算来保证隐私。

在研究方面，剪枝用于研究过参数化和欠参数化网络之间学习动力学的差异，研究幸运稀疏子网络和初始化的作用（“lottery tickets”）作为一种破坏性的神经结构搜索技术，等等。 在本教程中将学习如何使用 `torch.nn.utils.prune` 稀疏化神经网络，以及如何扩展它来实现定制的剪枝技术。

版本要求

`“torch>=1.4.0a0+8e8a5e0”`

In [2]:
```
import torch
from torch import nn
import torch.nn.utils.prune as prune
import torch.nn.functional as F
```

创建模型

在本教程中，使用LeNet网络架构。

In [3]:
```
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, 3x3 square conv kernel
        self.conv1 = nn.Conv2d(1, 6, 3)
        self.conv2 = nn.Conv2d(6, 16, 3)
        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)
```

检查模块

检查一下LeNet模型中的（未调整的） `conv1` 层。它将包含2个参数 `weight` 和 `bias` ，目前没有buffers。

In [30]:
```
module = model.conv1
print(list(module.named_parameters()))
```

```
[('weight', Parameter containing:
tensor([[[[ 0.2018,  0.1426, -0.2464],
          [-0.1109,  0.0992, -0.0712],
          [-0.1388,  0.1873,  0.0897]]],


        [[[-0.0597,  0.2355,  0.1157],
          [-0.0819,  0.1940,  0.1736],
          [-0.2958, -0.0601,  0.2093]]],


        [[[-0.0418,  0.0144,  0.3214],
          [-0.3300, -0.1674, -0.0756],
          [-0.1359,  0.2351, -0.0671]]],


        [[[-0.1044, -0.2492, -0.0118],
          [-0.1881,  0.0357, -0.3060],
          [-0.3047,  0.1266,  0.3164]]],


        [[[-0.0285,  0.0373,  0.0923],
          [-0.2350,  0.2356, -0.2605],
          [-0.0580, -0.3308,  0.1967]]],


        [[[-0.0973, -0.1890, -0.0954],
          [ 0.0044, -0.3117,  0.0092],
          [-0.2943, -0.2327,  0.2568]]]], requires_grad=True)), ('bias', Parameter containing:
tensor([ 0.0339,  0.1344,  0.0584, -0.2045,  0.1927,  0.0413],
       requires_grad=True))]
```

In [5]:
```
print(list(module.named_buffers()))
```

```
[]
```

剪枝模块

要剪枝模块（在本例中，是LeNet架构的 `conv1` 层），首先从 `torch.nn.utils.prune` （或implement中选择一种剪枝技术，通过子类化 `BasePruningMethod` ）。然后，指定模块和要在该模块中剪枝的参数的名称。最后，使用所选剪枝技术所需的足够的关键字参数，指定剪枝参数。

在这个例子中，将随机修剪 `conv1` 层中名为 `weight` 的参数中30%的连接。模块作为第一个参数传递给函数 `name` 使用其字符串标识符来标识该模块内的参数； `amount` 表示要剪枝的连接的百分比（如果它是0和1.之间的浮点值），或者要剪枝的连接的绝对数量（如果它是非负整数）。

In [6]:
```
prune.random_unstructured(module, name="weight", amount=0.3)
```

Out[6]:
```
Conv2d(1, 6, kernel_size=(3, 3), stride=(1, 1))
```

剪枝的作用是从参数中删除 `weight` ，并将其替换为一个名为 `weight_orig` 的新参数（即在初始参数 `name` 后附加 `_orig` ） `weight_orig` 存储张量的未编辑版本。 `bias` 没有被删除，因此它将保持原样。

In [7]:
```
print(list(module.named_parameters()))
```

```
[('bias', Parameter containing:
tensor([ 0.0850,  0.1487, -0.0256, -0.3024, -0.3094, -0.0063], device='cuda:0',
       requires_grad=True)), ('weight_orig', Parameter containing:
tensor([[[[ 0.0859, -0.2052, -0.3183],
          [-0.0472,  0.2256,  0.3241],
          [-0.2393,  0.1412,  0.3021]]],


        [[[-0.1574, -0.0305, -0.2204],
          [-0.2729,  0.0476,  0.1325],
          [-0.0042, -0.1854,  0.0132]]],


        [[[ 0.3159, -0.0525, -0.0692],
          [ 0.3329,  0.1598,  0.1786],
          [-0.1007, -0.0644,  0.0190]]],


        [[[-0.1127,  0.3125, -0.0714],
          [-0.0282,  0.0414, -0.3149],
          [ 0.2416, -0.1738, -0.2789]]],


        [[[ 0.1509, -0.2466,  0.0588],
          [ 0.1181, -0.3000, -0.2938],
          [ 0.1595, -0.1375,  0.2574]]],


        [[[ 0.0653, -0.2723,  0.1146],
          [-0.2355, -0.3076,  0.0241],
          [ 0.2234,  0.3311, -0.2366]]]], device='cuda:0', requires_grad=True))]
```

通过上面选择的修剪技术生成的修剪Mask被保存为名为 `weight_mask` 的模块缓冲器（即，将 `_mask` 附加到初始参数 `name` ）。

In [8]:
```
print(list(module.named_buffers()))
```

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


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


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


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


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


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

要使前向传递在不进行修改的情况下工作， `weight` 属性必须存在。在 `torch.nn.utils.prune` 中实现

的剪枝技术计算权重的修剪版本（通过将Mask与原始参数组合），并将它们存储在属性 `weight` 中。

请注意，这不再是 `module` 的参数，它现在只是一个属性。

In [9]:
```
print(module.weight)
```

```
tensor([[[[ 0.0859, -0.2052, -0.3183],
          [-0.0472,  0.0000,  0.0000],
          [-0.2393,  0.1412,  0.0000]]],


        [[[-0.1574, -0.0305, -0.0000],
          [-0.2729,  0.0476,  0.1325],
          [-0.0000, -0.0000,  0.0132]]],


        [[[ 0.3159, -0.0525, -0.0000],
          [ 0.3329,  0.1598,  0.1786],
          [-0.1007, -0.0000,  0.0190]]],


        [[[-0.0000,  0.3125, -0.0714],
          [-0.0282,  0.0414, -0.3149],
          [ 0.2416, -0.1738, -0.2789]]],


        [[[ 0.1509, -0.0000,  0.0000],
          [ 0.1181, -0.0000, -0.2938],
          [ 0.0000, -0.0000,  0.2574]]],


        [[[ 0.0653, -0.2723,  0.1146],
          [-0.0000, -0.3076,  0.0000],
          [ 0.2234,  0.3311, -0.2366]]]], device='cuda:0',
       grad_fn=<MulBackward0>)
```

最后，使用PyTorch的 `forward_pre_books` 在每次前向传递之前应用修剪。

具体地说，当 `module` 被修剪时，正如在这里所做的那样，它将为与它相关联的每个被修剪的参数获取一个 `forward_pre_book` 。在这种情况下，由于到目前为止只修剪了名为 `weight` 的原始参数，因此将只存在一个钩子。

In [10]:
```
print(module._forward_pre_hooks)
```

```
OrderedDict([(0, <torch.nn.utils.prune.RandomUnstructured object at 0x0000029485503988>)])
```

为了完整起见，现在也可以修剪 `bias` ，看看 `module` 的参数、缓冲区、钩子和属性是如何变化的。只是为了尝试另一种修剪技术，这里按 `L1范数` 修剪 `bias` 中的3个最小条目，正如在 `L1_structured` 修剪函数中实现的那样。

In [11]:
```
prune.l1_unstructured(module, name="bias", amount=3)
```

Out[11]:
```
Conv2d(1, 6, kernel_size=(3, 3), stride=(1, 1))
```

现在期望命名的参数包括 `weight_orig` （来自之前）和 `bias_orig` 。缓冲区将包括 `weight_mask` 和 `bias_mask` 。2个张量的修剪版本将作为模块属性存在，并且模块现在将具有2个 `forward_pre_books` 。

In [12]:
```
print(list(module.named_parameters()))
```

```
[('weight_orig', Parameter containing:
tensor([[[[ 0.0859, -0.2052, -0.3183],
          [-0.0472,  0.2256,  0.3241],
          [-0.2393,  0.1412,  0.3021]]],


        [[[-0.1574, -0.0305, -0.2204],
          [-0.2729,  0.0476,  0.1325],
          [-0.0042, -0.1854,  0.0132]]],


        [[[ 0.3159, -0.0525, -0.0692],
          [ 0.3329,  0.1598,  0.1786],
          [-0.1007, -0.0644,  0.0190]]],


        [[[-0.1127,  0.3125, -0.0714],
          [-0.0282,  0.0414, -0.3149],
          [ 0.2416, -0.1738, -0.2789]]],


        [[[ 0.1509, -0.2466,  0.0588],
          [ 0.1181, -0.3000, -0.2938],
          [ 0.1595, -0.1375,  0.2574]]],


        [[[ 0.0653, -0.2723,  0.1146],
          [-0.2355, -0.3076,  0.0241],
          [ 0.2234,  0.3311, -0.2366]]]], device='cuda:0', requires_grad=True)), ('bias_orig',
Parameter containing:
tensor([ 0.0850,  0.1487, -0.0256, -0.3024, -0.3094, -0.0063], device='cuda:0',
       requires_grad=True))]
```

In [13]:
```
print(list(module.named_buffers()))
```

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


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


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


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


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


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

In [14]:
```
print(module.bias)
```

```
tensor([ 0.0000,  0.1487, -0.0000, -0.3024, -0.3094, -0.0000], device='cuda:0',
       grad_fn=<MulBackward0>)
```

In [15]:
```
print(module._forward_pre_hooks)
```

```
OrderedDict([(0, <torch.nn.utils.prune.RandomUnstructured object at 0x0000029485503988>), (1, <
torch.nn.utils.prune.L1Unstructured object at 0x0000029485505C88>)])
```

迭代修剪

模块中的同一参数可以被修剪多次，各种修剪调用的效果等于串联应用的各种Mask的组合。

新Mask与旧Mask的组合由 `PruningContainer` 的 `compute_mask` 方法处理。

例如，现在想进一步修剪 `module.weight` ，这一次使用沿着张量的第0轴的结构化修剪（第0轴对应于卷积层的输出通道，对于 `conv1` 具有维度6），基于通道的 `L2范数` 。这可以使用 `ln_structed` 函数来实现，其中 `n=2` 和 `dim=0` 。

In [16]:
```
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)
```

```
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]]],


        [[[ 0.3159, -0.0525, -0.0000],
          [ 0.3329,  0.1598,  0.1786],
          [-0.1007, -0.0000,  0.0190]]],


        [[[-0.0000,  0.3125, -0.0714],
          [-0.0282,  0.0414, -0.3149],
          [ 0.2416, -0.1738, -0.2789]]],


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


        [[[ 0.0653, -0.2723,  0.1146],
          [-0.0000, -0.3076,  0.0000],
          [ 0.2234,  0.3311, -0.2366]]]], device='cuda:0',
       grad_fn=<MulBackward0>)
```

相应的钩子现在将是 `torch.nn.utils.prune.PrunningContainer` 类型，并将存储应用于 `weight` 参数的修剪。

In [17]:
```
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
```

```
[<torch.nn.utils.prune.RandomUnstructured object at 0x0000029485503988>, <torch.nn.utils.prune.LnStructured object at 0x0000029485528088>]
```

序列化修剪后的模型

所有相关张量，包括Mask缓冲区和用于计算修剪张量的原始参数，都存储在模型的 `state_dict` 中，因此如果需要，可以很容易地序列化和保存。

In [18]:
```
print(model.state_dict().keys())
```

```
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'])
```

删除修剪重新参数化

为了使修剪永久化，删除根据 `weight_orig` 和 `weight_mask` 进行的重新参数化，并删除 `forward_pre_book` ，可以使用 `torch.nn.utils.prune` 中的 `remove` 功能。

请注意，这并不能撤消修剪，就好像它从未发生过一样。相反，它只是通过将参数 `weight` 重新分配给修剪后的模型参数，使其永久化。

Prior to removing the re-parametrization:

In [19]:
```
print(list(module.named_parameters()))
```

```
[('weight_orig', Parameter containing:
tensor([[[[ 0.0859, -0.2052, -0.3183],
          [-0.0472,  0.2256,  0.3241],
          [-0.2393,  0.1412,  0.3021]]],


        [[[-0.1574, -0.0305, -0.2204],
          [-0.2729,  0.0476,  0.1325],
          [-0.0042, -0.1854,  0.0132]]],


        [[[ 0.3159, -0.0525, -0.0692],
          [ 0.3329,  0.1598,  0.1786],
          [-0.1007, -0.0644,  0.0190]]],


        [[[-0.1127,  0.3125, -0.0714],
          [-0.0282,  0.0414, -0.3149],
          [ 0.2416, -0.1738, -0.2789]]],


        [[[ 0.1509, -0.2466,  0.0588],
          [ 0.1181, -0.3000, -0.2938],
          [ 0.1595, -0.1375,  0.2574]]],


        [[[ 0.0653, -0.2723,  0.1146],
          [-0.2355, -0.3076,  0.0241],
          [ 0.2234,  0.3311, -0.2366]]]], device='cuda:0', requires_grad=True)), ('bias_orig', Parameter containing:
tensor([ 0.0850,  0.1487, -0.0256, -0.3024, -0.3094, -0.0063], device='cuda:0',
       requires_grad=True))]
```

In [20]:
```
print(list(module.named_buffers()))
```

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


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


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


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


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


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

In [21]:
```
print(module.weight)
```

```
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]]],


         [[[ 0.3159, -0.0525, -0.0000],
          [ 0.3329,  0.1598,  0.1786],
          [-0.1007, -0.0000,  0.0190]]],


         [[[-0.0000,  0.3125, -0.0714],
          [-0.0282,  0.0414, -0.3149],
          [ 0.2416, -0.1738, -0.2789]]],


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


         [[[ 0.0653, -0.2723,  0.1146],
          [-0.0000, -0.3076,  0.0000],
          [ 0.2234,  0.3311, -0.2366]]]], device='cuda:0',
       grad_fn=<MulBackward0>)
```

After removing the re-parametrization:

In [22]:
```
prune.remove(module, 'weight')
print(list(module.named_parameters()))
```

```
[('bias_orig', Parameter containing:
tensor([ 0.0850,  0.1487, -0.0256, -0.3024, -0.3094, -0.0063], 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]]],


        [[[ 0.3159, -0.0525, -0.0000],
          [ 0.3329,  0.1598,  0.1786],
          [-0.1007, -0.0000,  0.0190]]],


        [[[-0.0000,  0.3125, -0.0714],
          [-0.0282,  0.0414, -0.3149],
          [ 0.2416, -0.1738, -0.2789]]],


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


        [[[ 0.0653, -0.2723,  0.1146],
          [-0.0000, -0.3076,  0.0000],
          [ 0.2234,  0.3311, -0.2366]]]], device='cuda:0', requires_grad=True))]
```

In [23]:
```
print(list(module.named_buffers()))
```

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

修剪模型中的多个参数

通过指定所需的修剪技术和参数，可以很容易地修剪网络中的多个张量，可能是根据它们的类型，正如将在本例中看到的那样。

In [24]:
```
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
```

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

全局剪枝

到目前为止，只研究了通常被称为“局部”修剪的方法，即通过将每个条目的统计信息（权重大小、激活、梯度等）与该张量中的其他条目进行比较，逐个修剪模型中的张量。然而，一种常见的、也许更强大的技术是一次修剪模型，方法是删除（例如）整个模型中最低20%的连接，而不是删除每层中最低的20%的连接。这可能会导致每层的修剪百分比不同。

让我们看看如何使用 `torch.nn.utils.prune` 中的 `global_ustructured` 来实现这一点。

In [25]:
```
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,
)
```

现在可以检查每个修剪后的参数中引起的稀疏性，这在每一层中都不等于20%。然而，全局稀疏性将（大约）为20%。

In [26]:
```
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()
        )
    )
)
```

```
Sparsity in conv1.weight: 1.85%
Sparsity in conv2.weight: 8.80%
Sparsity in fc1.weight: 22.05%
Sparsity in fc2.weight: 12.11%
Sparsity in fc3.weight: 10.12%
Global sparsity: 20.00%
```

使用自定义修剪函数扩展 `torc.nn.utils.prune`

要实现自己的修剪函数，可以像所有其他修剪方法一样，通过子类化 `BasePruningMethod` 基类来扩展 `nn.utils.prune` 模块。该基类为您实现以下方法： `__call__` 、 `apply_mask` 、 `application` 、 `prune` 和 `remove` 。除了一些特殊情况外，不应该为新的修剪技术重新实现这些方法。

然而，必须实现 `__init__` （构造函数）和 `compute_mask` （关于如何根据修剪技术的逻辑计算给定张量的Mask的说明）。此外，必须指定此技术实现的修剪类型（支持的选项有 `'global'` 、 `'structured'` 和 `'unstructured'` ）。在迭代应用修剪的情况下，需要确定如何组合Mask。换句话说，当修剪预修剪的参数时，当前的修剪技术预计会作用于参数的未修剪部分。指定 `PRUNING_TYPE` 将使 `PruningContainer` （处理修剪Mask的迭代应用）能够正确识别要修剪的参数切片。

例如，假设您想要实现一种修剪技术，该技术修剪张量中的每一个其他条目（或者——如果张量之前已经修剪过——在张量的剩余未编辑部分）。这将是 `PRUNING_TYPE='unstructured'` 的，因为它作用于层中的单个连接，而不是整个单元/通道（“结构化”）或不同参数（“全局”）。

In [27]:
```
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
```

现在，要将其应用于 `nn.Module` 中的参数，还应该提供一个简单的函数来实例化方法并应用它。

In [28]:
```
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!

In [29]:
```
model = LeNet()
foobar_unstructured(model.fc3, name='bias')

print(model.fc3.bias_mask)
```

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

In [ ]:
```

```