Streamlining Tensor and Network Pruning in PyTorch

Workshop on Practical ML for Developing Countries, ICLR 2020
Michela Paganini (Facebook AI Research), Jessica Forde (Brown University)
What is pruning?

- Pruning methods selectively set weights of a neural network to zero, sparsifying the model.
- Pruned models can maintain the accuracy of the original model and gain computational efficiency for on-device use.
- Methods remove weights based on different heuristics, such as their absolute value.
- Weights can also be removed in a structured way, zero-ing out an entire channel, or in a unstructured manner.
- In PyTorch, pruning is performed through the application of a mask onto the parameter.
torch.nn.utils.prune

Different tensor pruning techniques enabled under a unified framework

BasePruningMethod

```
CLASS torch.nn.utils.prune.BasePruningMethod
    Abstract base class for creation of new pruning techniques.

    CLASSMETHOD apply(module, name, *args, **kwargs)

        apply_mask(module)

        ABSTRACT compute_mask(t, default_mask)

        prune(t, default_mask=none)

        remove(module)
```

New pruning technique?
Just subclass BasePruningMethod and implement compute_mask!
torch.nn.utils.prune implements the logic that defines which portions of the tensors will be zeroed out while accounting for previously pruned entries.

For example, in `prune.L1Unstructured`:

- Defines the interface → concrete subclasses must implement the logic.
- Implements logic for zeroing out weights based on magnitude.
- Handles case where tensor had previously been pruned by computing valid entries and applying new pruning technique exclusively on those entries.

Abstract base class for creation of new pruning techniques.
PruningContainer

PruningContainer()

[SomePruningMethod(), AnotherPruningMethod(), FinalPruningMethod()]

compute_mask(t)

masks[0] -> {cumulative_mask}

compute_mask(t[slice])

masks[1] -> {cumulative_mask}

compute_mask(t[slice][slice])

masks[2] -> {cumulative_mask}
torch.nn.utils.prune

BasePruningMethod

CLASS torch.nn.utils.prune.BasePruningMethod

Abstract base class for creation of new pruning techniques.

CLASSMETHOD apply(module, name, *args, **kwargs)

apply_mask(module)

ABSTRACT compute_mask(t, default_mask)

prune(t, default_mask=None)

del(module)

Reparametrizes the pruned tensor in terms of the original tensor and the pruning mask, and adds a forward pre-hook to enable pruning on the fly.

1. if there is any previous mask applied to this parameter
   1. fetch the previous mask
   2. combine successive pruning calls into a prune.PruningContainer
2. move the unpruned parameter to "<param_name>_orig"
3. compute new mask via compute_mask
4. add mask as a buffer named "<param_name>_mask"
5. attach the pruned version of the tensor as an attribute
6. register the pruning technique as a forward pre-hook

Before pruning

"weight" is an unpruned parameter

During apply

the unpruned parameter is moved to "weight_orig"

the mask is saved to a buffer called "weight_mask"

the pruned tensor is stored as an attribute called "weight"
torch.nn.utils.prune

**BasePruningMethod**

Abstract base class for creation of new pruning techniques.

- **CLASS** `torch.nn.utils.prune.BasePruningMethod` [SOURCE]
- **CLASSMETHOD** `apply(module, name, *args, **kwargs)` [SOURCE]
- `apply_mask(module)` [SOURCE]
- **ABSTRACT** `compute_mask(t, default_mask)` [SOURCE]
- `prune(t, default_mask=None)` [SOURCE]
- `remove(module)` [SOURCE]

Fetches the mask and the original, unpruned tensor to compute the pruned tensor during the forward pass — op is accounted for in the backward pass, too.

---

FACEBOOK AI
Makes the pruning reparametrization permanent

During `remove`

- The pruned tensor is moved to a parameter called "weight"
- "weight_orig" and "weight_mask" are permanently deleted

After pruning

- The unpruned parameter is stored in "weight_orig"
- The mask is stored as a buffer in "weight_mask"
- The pruned tensor is stored as an attribute in "weight"
torch.nn.utils.prune is designed to act on a torch.nn.Module.
**torch.nn.utils.prune**

**Easy to use**

```python
def LeNet() -> torch.nn.Module:
    return ...

model = LeNet()  # unpruned model

# L_2 structured pruning will remove 50% of channels across axis 0
prune.ln_structured(
    module=model.conv1,
    name="weight",
    amount=0.5,
    n=2,
    dim=0
)
```

**Iterative pruning made easy**

```python
for _ in range(10):
    # Remove 2 connections per iteration
    prune.l1_unstructured(module=model.fc1, name="bias", amount=2)
```

**Global pruning made easy**

```python
parameters_to_prune = (  
    (model.conv1, "weight"),  
    (model.conv2, "weight"),  
    (model.fc1, "weight"),  
)

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

**Easy to extend**

**supports 3 PRUNING_TYPES: 'global', 'structured', and 'unstructured' (to determine how to combine masks if pruning is applied iteratively)**

**instructions on how to compute the mask for the given tensor according to the logic of your pruning technique**

```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

def fooobar_unstructured(module, name):
    FooBarPruningMethod.apply(module, name)
    return module
```
GlobalPruning

torch.nn.utils.prune.global_unstructured(...)

layer 1
layer 2
layer 3
Questions?

Contact: michela@fb.com, jessica_forde@brown.edu