Intro

The Base Pruner inherits from the Base Sparsifier.

Motivation

Sparsifying weights allows us to skip some of the multiplications during the dot product (i.e. in the Linear layers), which ultimately translates into faster inference. With structured pruning, whole rows/columns of a tensor would be zeroed-out. This translates into model transformation (not just tensor transformation). Logically, the process of structured pruning is similar to removing some of the input/output channels in the layer completely.

prune logic

Design Choices

Eager Mode

PruningParametrization: After pruning, the shape of the weight changes (some of the output channels are pruned). That means the output of the current layer will have less output layers compared to the original. This means that the next layer should have less input channels.

Consider an example below:

prune example

The dot product of the masked matrix A (weight) and matrix B (activation) produces the zeros at the sparse locations. However, if we remove the zeros, as in the example shown earlier, the result will change:

prune result

The resulting matrix is of different shape (2x2 vs. 4x2).

**Forward Hook - ActivationReconstruction **(aka re-inserting zeros): To reconstruct the activation with the original shape, we will undo the sparsification before pushing that activation to the next layer. We do this with a forward hook -- forward hooks are functions that are called on the activation after the computation is complete.

prune reconstruction

Forward Hook - Bias:

If the layer has a bias, it must be added to the activation AFTER zeros have been re-inserted, i.e. after the `ActivationReconstruction` forward hook.

The pruner prunes the entire channel by default (weight & corresponding bias), so indices of the bias corresponding to pruned indices will be zeroed out.

Eager Mode APIs & Code Snippets

Supported modules: nn.Linear, nn.Conv2d, nn.BatchNorm2d*

- when provided in `config` with corresponding Conv2d layer

`BasePruner` : base class with abstract method `update_mask` that computes the new pruner mask for all modules (see Write Your Own Pruner). The base pruner prunes the entire channel by default (weight & corresponding bias); if you don't want the bias to be pruned, then set `also_prune_bias` to be False.

`prepare` : registers the pruning parametrization (called `PruningParametrization`) to each module layer of the model; also adds forward hooks for bias support and re-inserting zeros to the output so the next layer received the correct size input.

Note: for BatchNorm2d layers, the parametrization `ZeroesParametrization` is attached instead since its weight is 1d, so removing channels would affect the input dimension as well. `ZeroesParametrization` zeroes out channels rather than removing them like `PruningParametrization` . We need this when `also_prune_bias=True` , so BatchNorm2d channels get pruned with their corresponding Conv2d channels.

```
pruner = ImplementedPruner(defaults=None, also_prune_bias=True)
pruner.prepare(model, config)
```

`step` : applies `update_mask` logic (i.e. prunes the weight matrix)

```
pruner.step()
```

`squash_mask` : applies the parametrization one last time to the weight matrix, and then removes the pruning parametrization from the model

```
pruner.squash_mask()
```

Write Your Own Pruner

To write a custom pruner, one could inherit from the `BasePruner` and implement some of the methods. For example, if implementing a pruner that computes the mask by randomly pruning ⅓ of channels:

```
class ImplementedPruner(BasePruner):
   def update_mask(self, layer, **kwargs):
       param = layer.parametrizations.weight[0]  # PruningParametrization
       all_outputs = param.original_outputs
       prune = random.sample(all_outputs, len(all_outputs) // 3)
       param.pruned_outputs.update(prune)
```

It is the responsibility of the base class to call the `self.update_mask` when appropriate.