HW5

April 21, 2021

1 Homework 5

Starting from the implementation contained within the notebook `05-pruning.ipynb`, extend the `magnitude_pruning` function to allow for incremental (iterative) pruning. In the current case, if you try pruning one more time, you'll notice that it will not work as there's no way to communicate to the future calls of `magnitude_pruning` to ignore the parameters which have already been pruned. Find a way to enhance the routine s.t. it can effectively prune networks in a sequential fashion (i.e., if we passed an MLP already pruned of 20% of its parameters, we want to prune *another* 20% of parameters).

First, import all libraries and modules needed.

[146]:
```
import torch
from scripts import mnist, train_utils, architectures, train
from scripts.train_utils import accuracy, AverageMeter
from scripts.torch_utils import use_gpu_if_possible
```

My improved `magnitude_pruning` function:

[147]:
```
def magnitude_pruning(model, pruning_rate, layers_to_prune, mask=None):
    # mask is the starting mask for the process

    if mask is None:

        params_to_prune = [pars[1] for pars in model.named_parameters() if␣
↪any([l in pars[0] for l in layers_to_prune])]
        flat = torch.cat([pars.abs().flatten() for pars in params_to_prune],␣
↪dim=0)

        flat = flat.sort()[0]

        position = int(pruning_rate * flat.shape[0])
        thresh = flat[position]

        mask = []
        for pars in model.named_parameters():
            if any([l in pars[0] for l in layers_to_prune]):
                m = torch.where(pars[1].abs() >= thresh, 1, 0)
```

```
                mask.append(m)
                pars[1].data *= m
            else:
                mask.append(torch.ones_like(pars[1]))

        return mask

    else:
        params_to_prune = [m*params for (name, params),m in zip(model.
↪named_parameters(), mask)
                           if any([layer in name for layer in␣
↪layers_to_prune])]
        flat = torch.cat([pars.abs().flatten() for pars in params_to_prune],␣
↪dim=0)
        flat = flat.sort()[0]
        flat = flat[flat.nonzero()]
        position = int(pruning_rate* flat.shape[0])
        thresh = flat[position]

        new_mask = []
        for i, ((name, param),m) in enumerate(zip(model.named_parameters(),␣
↪mask)):
            if any([layer in name for layer in layers_to_prune]):
                new_m = torch.where(m*param.abs() >= thresh, 1, 0)
                new_mask.append(new_m)
                param.data *= new_m
            else:
                new_mask.append(torch.ones_like(param))

        return new_mask
```

Let's see if it works. From the provided notebook `05-pruning.ipynb`:

[148]:
```
def train_epoch(model, dataloader, loss_fn, optimizer, loss_meter,␣
↪performance_meter, performance, device, mask, layers_to_prune,␣
↪params_type_to_prune):
    for X, y in dataloader:
        X = X.to(device)
        y = y.to(device)

        optimizer.zero_grad()
        y_hat = model(X)
        loss = loss_fn(y_hat, y)
        loss.backward()

        if mask is not None:
            for (name, param), m in zip(model.named_parameters(), mask):
```

```
                if any([l in name for l in layers_to_prune]):
                    param.grad *= m

        optimizer.step()
        acc = performance(y_hat, y)

        loss_meter.update(val=loss.item(), n=X.shape[0])
        performance_meter.update(val=acc, n=X.shape[0])
```

[149]:

```
def train_model(model, dataloader, loss_fn, optimizer, num_epochs,␣
↪checkpoint_loc=None, checkpoint_name="checkpoint.pt", performance=accuracy,␣
↪lr_scheduler=None, device=None, mask=None, layers_to_prune=None,␣
↪params_type_to_prune=["weight", "bias"]):
    if checkpoint_loc is not None:
        os.makedirs(checkpoint_loc, exist_ok=True)

    if device is None:
        device = use_gpu_if_possible()

    model = model.to(device)
    model.train()

    for epoch in range(num_epochs):
        loss_meter = AverageMeter()
        performance_meter = AverageMeter()

        print(f"Epoch {epoch+1} --- learning rate {optimizer.
↪param_groups[0]['lr']:.5f}")

        train_epoch(model, dataloader, loss_fn, optimizer, loss_meter,␣
↪performance_meter, performance, device, mask, layers_to_prune,␣
↪params_type_to_prune)

        print(f"Epoch {epoch+1} completed. Loss - total: {loss_meter.sum} -␣
↪average: {loss_meter.avg}; Performance: {performance_meter.avg}")

        if checkpoint_name is not None and checkpoint_loc is not None:
            checkpoint_dict = {
                "parameters": model.state_dict(),
                "optimizer": optimizer.state_dict(),
                "epoch": epoch
            }
            torch.save(checkpoint_dict, os.path.join(checkpoint_loc,␣
↪checkpoint_name))

        if lr_scheduler is not None:
            lr_scheduler.step()
```

```
    return loss_meter.sum, performance_meter.avg
```

[150]:
```
layers = [
    {"n_in": 784, "n_out": 16, "batchnorm": False},
    {"n_out": 32, "batchnorm": True},
    {"n_out": 64, "batchnorm": True},
    {"n_out": 10, "batchnorm": True}
]
net = architectures.MLPCustom(layers)
print(net)
```

```
MLPCustom(
  (layers): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=784, out_features=16, bias=True)
    (2): ReLU()
    (3): BatchNorm1d(16, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (4): Linear(in_features=16, out_features=32, bias=True)
    (5): ReLU()
    (6): BatchNorm1d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (7): Linear(in_features=32, out_features=64, bias=True)
    (8): ReLU()
    (9): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (10): Linear(in_features=64, out_features=10, bias=True)
    (11): ReLU()
  )
)
```

[151]:
```
def number_of_ones_in_mask(mask):
    return sum([m.sum().item() for m in mask]) / sum([m.numel() for m in mask])
```

Iterative pruning:

[152]:
```
trainloader, testloader, _, _ = mnist.get_data()
loss_fn = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(net.parameters(), lr=0.1)
```

[153]:
```
train_model(net, trainloader, loss_fn, optimizer, num_epochs=3,␣
↪layers_to_prune=["1", "4", "7", "10"])
```

```
Epoch 1 --- learning rate 0.10000
Epoch 1 completed. Loss - total: 24872.93850517273 - average:
0.41454897508621213; Performance: 0.8872833333333333
Epoch 2 --- learning rate 0.10000
```

```
Epoch 2 completed. Loss - total: 12579.992371559143 - average:
0.2096665395259857; Performance: 0.939
Epoch 3 --- learning rate 0.10000
Epoch 3 completed. Loss - total: 10631.141635417938 - average:
0.17718569392363231; Performance: 0.9486
```

[153]: (10631.141635417938, 0.9486)

[154]:
```
mask = magnitude_pruning(net, 0.2, set(["1", "4", "7", "10"]))
print("Number of ones in mask:", number_of_ones_in_mask(mask), "\n")
```

```
Number of ones in mask: 0.8027967681789931
```

[155]:
```
train_model(net, trainloader, loss_fn, optimizer, num_epochs=3,␣
↪layers_to_prune=["1", "4", "7", "10"], mask=mask)
```

```
Epoch 1 --- learning rate 0.10000
Epoch 1 completed. Loss - total: 9261.125999450684 - average:
0.15435209999084473; Performance: 0.9545833333333333
Epoch 2 --- learning rate 0.10000
Epoch 2 completed. Loss - total: 8554.511769771576 - average:
0.1425751961628596; Performance: 0.9583166666666667
Epoch 3 --- learning rate 0.10000
Epoch 3 completed. Loss - total: 8190.981409549713 - average:
0.13651635682582855; Performance: 0.9587833333333333
```

[155]: (8190.981409549713, 0.9587833333333333)

[156]:
```
mask = magnitude_pruning(net, 0.2, set(["1", "4", "7", "10"]), mask=mask)
print("Number of ones in mask:", number_of_ones_in_mask(mask), "\n")
```

```
Number of ones in mask: 0.6450590428837788
```

[157]:
```
train_model(net, trainloader, loss_fn, optimizer, num_epochs=3,␣
↪layers_to_prune=["1", "4", "7", "10"], mask=mask)
```

```
Epoch 1 --- learning rate 0.10000
Epoch 1 completed. Loss - total: 7544.220559358597 - average:
0.12573700932264328; Performance: 0.9625166666666667
Epoch 2 --- learning rate 0.10000
Epoch 2 completed. Loss - total: 7218.545625925064 - average:
0.12030909376541774; Performance: 0.9628
Epoch 3 --- learning rate 0.10000
Epoch 3 completed. Loss - total: 7017.527235031128 - average:
0.1169587872505188; Performance: 0.965
```

[157]: (7017.527235031128, 0.965)

[158]:
```
mask = magnitude_pruning(net, 0.2, set(["1", "4", "7", "10"]),  mask=mask)
print("Number of ones in mask:", number_of_ones_in_mask(mask), "\n")
```

```
Number of ones in mask: 0.5188315724052206
```

Conclusion: The number of ones is the mask is reduced by 20% at every iteration, so it works.