Introduction to Deep Learning (CS474)

Lecture 18

Rupam

Outline

- **Module 2**

    - Discussion on building a simple ConvNet using PyTorch

Rupam

Recap

```
import torch
import torch.nn as nn

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)
        self.act1 = nn.Tanh()
        self.pool1 = nn.MaxPool2d(2)
        self.conv2 = nn.Conv2d(16, 8, kernel_size=3, padding=1)
        self.act2 = nn.Tanh()
        self.pool2 = nn.MaxPool2d(2)
        self.fc1 = nn.Linear(8 * 8 * 8, 32)
        self.act3 = nn.Tanh()
        self.fc2 = nn.Linear(32, 2)

    def forward(self, x):
        out = self.pool1(self.act1(self.conv1(x)))
        out = self.pool2(self.act2(self.conv2(out)))
        out = out.view(-1, 8 * 8 * 8) # <1>
        out = self.act3(self.fc1(out))
        out = self.fc2(out)
        return out
```

Slide credit: E. STEVENS, L. ANTIGA, and T. VIEHMANN

Parameters

```
model = Net()

numel_list = [p.numel() for p in model.parameters()]
sum(numel_list), numel_list
```

- No matter how nested the submodule, any `nn.Module` can access the list of all child parameters.

- By accessing their `grad` attribute, which has been populated by **autograd** , the optimizer will know how to change parameters to minimize the loss.

Slide credit: E. STEVENS, L. ANTIGA, and T. VIEHMANN

The *functional* API

```python
import torch.nn.functional as F
import torch
import torch.nn as nn

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(16, 8, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(8 * 8 * 8, 32)
        self.fc2 = nn.Linear(32, 2)

    def forward(self, x):
        out = F.max_pool2d(torch.tanh(self.conv1(x)), 2)
        out = F.max_pool2d(torch.tanh(self.conv2(out)), 2)
        out = out.view(-1, 8 * 8 * 8)
        out = torch.tanh(self.fc1(out))
        out = self.fc2(out)
        return out
```

Slide credit: E. STEVENS, L. ANTIGA, and T. VIEHMANN

Rupam

The *functional* API

- Indeed, `torch.nn.functional` provides many functions that work like the modules we find in nn .
- But instead of working on the input arguments and stored parameters like the module counterparts, they take inputs and parameters as arguments to the function call.
- For instance, the functional counterpart of `nn.Linear` is `nn.functional.linear` , which is a function that has signature `linear(input, weight, bias=None)` .
- Back to our model, it makes sense to keep using nn modules for nn.Linear and nn.Conv2d so that **Net** will be able to manage their Parameter s during training.
- However, we can safely switch to the functional counterparts of pooling and activation.

Slide credit: E. STEVENS, L. ANTIGA, and T. VIEHMANN

Checking Information Flow

- In order to check that the information in our model flows correctly, we need to **update** our notebook so that it can process CIFAR-10.

- Hope you remember our earlier classes!

```
import torch.nn.functional as F
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms

transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)

trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)

testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                         shuffle=False, num_workers=2)
```

Slide credit: E. STEVENS, L. ANTIGA, and T. VIEHMANN

Rupam

Checking Information Flow

- Hope you remember our earlier classes!

```
classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

label_map = {0: 0, 2: 1}
class_names = ['airplane', 'bird']
cifar2 = [(img, label_map[label])
          for img, label in trainset
          if label in [0, 2]]
cifar2_val = [(img, label_map[label])
              for img, label in testset
              if label in [0, 2]]
```

Slide credit: E. STEVENS, L. ANTIGA, and T. VIEHMANN

Rupam

Checking Information Flow

- Once you add above content, you should include **Net** !
- Now, we are in a position to check the information flow.

```
img, _ = cifar2[0]

model = Net()
model(img.unsqueeze(0))
```

```
Files already downloaded and verified
Files already downloaded and verified
tensor([[0.0681, 0.2275]], grad_fn=<AddmmBackward>)
```

Slide credit: E. STEVENS, L. ANTIGA, and T. VIEHMANN

Rupam

Training our *convnet*

```
import datetime
def training_loop(n_epochs, optimizer, model, loss_fn, train_loader):
    for epoch in range(1, n_epochs + 1):
        loss_train = 0.0
        for imgs, labels in train_loader:

            outputs = model(imgs)

            loss = loss_fn(outputs, labels)

            optimizer.zero_grad()

            loss.backward()

            optimizer.step()

            loss_train += loss.item()

        if epoch == 1 or epoch % 10 == 0:
            print('{} Epoch {}, Training loss {}'.format(
                datetime.datetime.now(), epoch,
                loss_train / len(train_loader)))
```

Slide credit: E. STEVENS, L. ANTIGA, and T. VIEHMANN

Rupam

Training our *convnet*

```
train_loader = torch.utils.data.DataLoader(cifar2, batch_size=64,
                                           shuffle=True)
model = Net()
optimizer = optim.SGD(model.parameters(), lr=1e-2)
loss_fn = nn.CrossEntropyLoss()

training_loop(  # <5>
    n_epochs = 100,
    optimizer = optimizer,
    model = model,
    loss_fn = loss_fn,
    train_loader = train_loader
)
```

Slide credit: E. STEVENS, L. ANTIGA, and T. VIEHMANN

Training our *convnet*

```
Files already downloaded and verified
Files already downloaded and verified
2020-10-20 04:34:55.160428 Epoch 1, Training loss 0.5823479412467616
2020-10-20 04:35:33.198386 Epoch 10, Training loss 0.37229365608861
2020-10-20 04:36:15.374541 Epoch 20, Training loss 0.31976451131568595
2020-10-20 04:36:57.928711 Epoch 30, Training loss 0.29985179386700794
2020-10-20 04:37:40.245963 Epoch 40, Training loss 0.2837665013636753
2020-10-20 04:38:22.353324 Epoch 50, Training loss 0.2666208679034452
2020-10-20 04:39:04.426601 Epoch 60, Training loss 0.25153562284199293
```

Slide credit: E. STEVENS, L. ANTIGA, and T. VIEHMANN

References

- All the contents present in the slides are taken from various online resources. Due credit is given in the respective slides.These slides are used for *academic* purposes only.