Deep learning

8.5. DataLoader and neuro-surgery

François Fleuret

https://fleuret.org/dlc/

UNIVERSITÉ DE GENÈVE

`torch.utils.data.DataLoader`

Until now, we have dealt with image sets that could fit in memory, and we manipulated them as regular tensors, e.g.

```
train_set = torchvision.datasets.MNIST(root = data_dir,
                                       train = True, download = True)
train_input = train_set.data.view(-1, 1, 28, 28).float()
train_targets = train_set.targets
```

Until now, we have dealt with image sets that could fit in memory, and we manipulated them as regular tensors, e.g.

```
train_set = torchvision.datasets.MNIST(root = data_dir,
                                       train = True, download = True)
train_input = train_set.data.view(-1, 1, 28, 28).float()
train_targets = train_set.targets
```

However, large sets do not fit in memory, and samples have to be constantly loaded during training.

ImageNet LSVRC 2012	Images	151Gb
LSUN (all classes)	Images	1.7Tb
OSCAR	Text	6Tb

This requires a [sophisticated] machinery to parallelize the loading itself, but also the normalization, and data-augmentation operations.

PyTorch offers the `torch.utils.data.DataLoader` object which combines a **data-set** and a **sampling policy** to create an iterator over mini-batches.

Standard data-sets are available in `torchvision.datasets`, and they allow to apply transformations over the images or the labels transparently.

PyTorch offers the `torch.utils.data.DataLoader` object which combines a **data-set** and a **sampling policy** to create an iterator over mini-batches.

Standard data-sets are available in `torchvision.datasets`, and they allow to apply transformations over the images or the labels transparently.

If needed, `torchvision.datasets.ImageFolder` creates a data-set from files located in a folder, and `torch.utils.data.TensorDataset` from a tensor. The latter is useful for synthetic toy examples or small data-sets.

```
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

data_dir = os.environ.get('PYTORCH_DATA_DIR') or './data/mnist/'

train_transforms = transforms.Compose(
    [
        transforms.ToTensor(),
        transforms.Normalize(mean = (0.1302,), std = (0.3069, ))
    ]
)

train_loader = DataLoader(
    datasets.MNIST(root = data_dir, train = True, download = True,
                   transform = train_transforms),
    batch_size = 100,
    num_workers = 4,
    shuffle = True,
    pin_memory = torch.cuda.is_available()
)
```

Given this `train_loader`, we can now re-write our training procedure with a loop over the mini-batches

```
for e in range(nb_epochs):
    for input, targets in iter(train_loader):

        input, targets = input.to(device), targets.to(device)

        output = model(input)
        loss = criterion(output, targets)

        model.zero_grad()
        loss.backward()
        optimizer.step()
```

Example of neuro-surgery and fine-tuning in PyTorch

As an example of re-using a network and fine-tuning it, we will construct a network for CIFAR10 composed of:

- the first layer of an [already trained] AlexNet,

As an example of re-using a network and fine-tuning it, we will construct a network for CIFAR10 composed of:

- the first layer of an [already trained] AlexNet,
- several resnet blocks,

As an example of re-using a network and fine-tuning it, we will construct a network for CIFAR10 composed of:

- the first layer of an [already trained] AlexNet,
- several resnet blocks,
- a final channel-wise averaging, using `nn.AvgPool2d`, and

As an example of re-using a network and fine-tuning it, we will construct a network for CIFAR10 composed of:

- the first layer of an [already trained] AlexNet,
- several resnet blocks,
- a final channel-wise averaging, using `nn.AvgPool2d`, and
- a final fully connected linear layer `nn.Linear`.

As an example of re-using a network and fine-tuning it, we will construct a network for CIFAR10 composed of:

- the first layer of an [already trained] AlexNet,
- several resnet blocks,
- a final channel-wise averaging, using `nn.AvgPool2d`, and
- a final fully connected linear layer `nn.Linear`.

During training, we will keep the AlexNet features frozen for a few epochs. This is done by setting `requires_grad` of the related `Parameter`s to `False`.

```
data_dir = os.environ.get('PYTORCH_DATA_DIR') or './data/cifar10/'

num_workers = 4
batch_size = 64

transform = torchvision.transforms.ToTensor()

train_set = datasets.CIFAR10(root = data_dir, train = True,
                             download = True, transform = transform)

train_loader = utils.data.DataLoader(train_set, batch_size = batch_size,
                                     shuffle = True, num_workers = num_workers)

test_set = datasets.CIFAR10(root = data_dir, train = False,
                            download = True, transform = transform)

test_loader = utils.data.DataLoader(test_set, batch_size = batch_size,
                                    shuffle = False, num_workers = num_workers)
```

```python
class ResBlock(nn.Module):
    def __init__(self, nb_channels, kernel_size):
        super().__init__()

        self.conv1 = nn.Conv2d(nb_channels, nb_channels, kernel_size,
                               padding = (kernel_size-1)//2)
        self.bn1 = nn.BatchNorm2d(nb_channels)

        self.conv2 = nn.Conv2d(nb_channels, nb_channels, kernel_size,
                               padding = (kernel_size-1)//2)
        self.bn2 = nn.BatchNorm2d(nb_channels)

    def forward(self, x):
        y = self.bn1(self.conv1(x))
        y = F.relu(y)
        y = self.bn2(self.conv2(y))
        y += x
        y = F.relu(y)
        return y
```

```
class Monster(nn.Module):
    def __init__(self, nb_blocks, nb_channels):
        super().__init__()

        alexnet = torchvision.models.alexnet(weights = 'IMAGENET1K_V1')

        self.features = nn.Sequential(alexnet.features[0], nn.ReLU(inplace = True))

        dummy = self.features(torch.zeros(1, 3, 32, 32)).size()
        alexnet_nb_channels = dummy[1]
        alexnet_map_size = tuple(dummy[2:4])

        self.conv = nn.Conv2d(alexnet_nb_channels, nb_channels, kernel_size = 1)

        self.resblocks = nn.Sequential(
            *(ResBlock(nb_channels, kernel_size = 3) for _ in range(nb_blocks))
        )

        self.avg = nn.AvgPool2d(kernel_size = alexnet_map_size)
        self.fc = nn.Linear(nb_channels, 10)
```

```python
def forward(self, x):
    x = self.features(x)
    x = F.relu(self.conv(x))
    x = self.resblocks(x)
    x = F.relu(self.avg(x))
    x = x.view(x.size(0), -1)
    x = self.fc(x)
    return x
```

```
nb_epochs = 50
nb_blocks, nb_channels = 8, 64

model, criterion = Monster(nb_blocks, nb_channels), nn.CrossEntropyLoss()

model.to(device)
criterion.to(device)

optimizer = torch.optim.Adam(model.parameters(), lr = 1e-2)

for e in range(nb_epochs):
    # Freeze the features during half of the epochs
    for p in model.features.parameters():
        p.requires_grad = e >= nb_epochs // 2

    acc_loss = 0.0

    for input, targets in iter(train_loader):
        input, targets = input.to(device), targets.to(device)

        output = model(input)
        loss = criterion(output, targets)
        acc_loss += loss.item()

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print(e, acc_loss)
```

```python
nb_test_errors, nb_test_samples = 0, 0

model.eval()

for input, targets in iter(test_loader):
    input, targets = input.to(device), targets.to(device)

    output = model(input)
    wta = torch.argmax(output.data, 1).view(-1)

    for i in range(targets.size(0)):
        nb_test_samples += 1
        if wta[i] != targets[i]: nb_test_errors += 1

test_error = 100 * nb_test_errors / nb_test_samples
print(f'test_error {test_error:.02f}% ({nb_test_errors}/{nb_test_samples})')
```

The end