3.3. Numerical optimization with pytorch

Gradient descent can write code better than you. I'm sorry.

Andrej Karpathy (full quote on page 1)

PyTorch is a library for numerical optimization. It's oriented around the types of optimization problems that arise in neural networks—the library routines are set up to make these problems easy to implement—but it can be used for any optimization at all.

This section will show some typical PyTorch optimization code and explain its conventions, so that you can make sense of the PyTorch code snippets used in this book. For information about how PyTorch works under the hood, or for guidelines about the software engineering aspects (working with large datasets, GPUs, clusters) you should look elsewhere.

MODULES, PARAMETERS, AND OPTIMIZERS

The core concepts in PyTorch code are modules, parameters, and optimizers. To illustrate them, here's an example.

Example 3.3.1.
Given a dataset $(x_1, y_1), \dots, (x_n, y_n)$ of points in $\mathbb{R}^2$, fit a simple straight-line linear regression

$$Y_i \sim \alpha + \beta x_i + N(0, \sigma^2).$$

This requires maximizing

$$\sum_{i=1}^{n} \Bigl\{ -\frac{1}{2} \log(2\pi\sigma^2) - \frac{1}{2\sigma^2} \bigl(y_i - f(x_i)\bigr)^2 \Bigr\} \quad \text{where} \quad f(x) = \alpha + \beta x$$

over the parameters α, β, and σ.

```
import torch
import torch.nn as nn
import torch.optim as optim

class StraightLine(nn.Module):
    def __init__(self):
        super().__init__()
        self.α = nn.Parameter(torch.tensor(1.0))
        self.β = nn.Parameter(torch.tensor(1.0))
    def forward(self, x):
        return self.α + self.β * x

class Y(nn.Module):
    def __init__(self):
        super().__init__()
        self.f = StraightLine()
        self.σ = nn.Parameter(torch.tensor(1.0))
    def forward(self, x, y):
        pred = self.f(x)
        return -0.5*torch.log(2*np.pi*self.σ**2) - (y-pred)**2/2/self.σ**2

model = Y()
optimizer = optim.Adam(model.parameters())
epoch = 0

with Interruptable() as check_interrupted:
    check_interrupted()
    optimizer.zero_grad()
    loglik = model(x, y)
    e = - torch.mean(loglik)
    e.backward()
    optimizer.step()
```

```
33        IPython.display.clear_output(wait=True)
34        print(f'epoch={epoch} loglik={-e.item():.3}')
35        epoch += 1
```

Modules. The class nn.Module represents a function, typically a parameterized function. In this code there are two functions, $f(x\,;\,\alpha,\beta)$ implemented in class StraightLine and $\Pr_Y(y\,;\,\alpha,\beta,\sigma)$ implemented in class Y. Each module should have a forward method that actually evaluates the function. When we call a module object directly, as in

```
model = Y()
model.f(3.0)
model(3.0, 1.0)
```

then it invokes forward. A module is a regular Python class, so we're free to define whatever other methods we like. This code could just as well have been written as a single module, perhaps with an extra method for computing f; but it's useful to see how PyTorch lets us split a big problem into smaller reusable building blocks.

Batched functions. There are many built-in modules in torch.nn. They nearly all have the convention that they operate on batches of records, where each item in the batch is to be operated on separately. It's a good idea to follow this convention. So, for example, if we want to implement a function that operates on 2d matrices, we actually make it operate on 3d arrays where the first dimension indexes the items in the batch. In this code, StraightLine and Y follow this convention: mathematically we think of them as applying to scalars, but the code works on vectors and it will act separately on each item in the vector.

Parameters. Any parameters that we'll want to optimize over, we declare in the module's __init__ with a nn.Parameter wrapper. The wrapper lets PyTorch keep track of parameters, so that when we set up the optimization loop on line 23 it's easy to specify what parameters need to be optimized. On line 16, Y uses a StraightLine object, which tells PyTorch that Y depends not just on its own parameter σ but also on the parameters from StraightLine. (We can also define whatever other fields we like in __init__, and if they don't have the Parameter wrapper then PyTorch will leave them alone.)

Optimization. For many optimization problems in machine learning, we iterate towards a solution, there aren't hard and fast stopping criteria, and it's useful to interrupt and inspect and perhaps tweak and resume the iteration. I like to run my PyTorch optimizations interactively, using the construction on line 26. (The Interruptable class is the author's, and it's given in the appendix.) This makes it easy to log the progress of the optimization, and interrupt it cleanly with Jupyter's Kernel | Interrupt menu item.

Interruptable class: appendix C page 205

PyTorch comes with several iterative optimization algorithms, all of them variations on gradient descent. This code selects the Adam optimizer on line 23, and tells it to optimize over all three parameters in the model. If you find yourself needing to meddle with the optimizer then you're in the domain of engineering, not modelling, and that's outside the scope of this book.

PyTorch optimizers all seek a minimum. We want to maximize the log likelihood of the dataset, i.e. minimize the negative log likelihood, which is what line 30 specifies.

Tensors and the computation graph. Anything numerical we do in PyTorch, we do on torch.tensor objects. These are similar to numpy arrays. There are tensor methods that mimic many of the numpy routines: torch.zeros, torch.log, etc.

But tensors are richer than numpy arrays, because they allow PyTorch to keep track of the *computation graph* of which operations were performed on which tensors. It uses this graph for its gradient descent calculations. You don't need to know anything about computation graphs for basic PyTorch use, but you do need to know about permitted operations ...

- All the operations in the functions you're optimizing should be PyTorch functions: torch.log, etc. You're not allowed to take values out of torch tensors, feed them into some other package, get the answers, and put them back into your module. If you did that, PyTorch wouldn't know how to apply gradient descent.
- If you want to inspect a value e.g. to plot it, use x.detach().numpy(), or if the value is a scalar use x.item(). This extracts the values from the PyTorch universe, and it will no longer be able to keep track of the computation graph for the operations you're performing. If you do want to

do computations with PyTorch tensors rather than numpy, wrap them in with torch.no_grad() to tell PyTorch that this is 'dead end' code and it's not to build a computation graph.
- If you want to reset the value of a parameter, use x.data=.... Don't do it inside optimization loop between optimizer.zero_grad() and optimizer.step(), since that might mess up the computation graph.

TRAINING A NEURAL NETWORK

Training a neural network involves exactly the same sort of code as above, with modules, parameters, and optimizers. Here's an example.

Example 3.3.2 (MNIST digit classification).
Consider the MNIST image dataset described on page 47, consisting of pairs (x_i, y_i) where $x_i \in \mathbb{R}^{28\times 28}$ is an image and $y_i \in \{0, 1, \dots, 9\}$ is its label. Consider the general probability model described on page 50, in which $\vec{f} : x \mapsto \mathbb{R}^{10}$ is some neural network that outputs a vector of scores, a score for each possible digit, and we model the label as

$$Y_i \sim \mathrm{Cat}\big(\mathrm{softmax}(\vec{f}(x_i))\big).$$

Training the neural network consists in maximizing the log likelihood of the dataset,

$$\log \Pr(y_1, \dots, y_n) = \sum_{i=1}^{n} \log\big[\mathrm{softmax}\big(\vec{f}(x_i)\big)\big]_{y_i}.$$

The dataset can be loaded with the following code, which returns a list $[(x_1, y_1), \dots, (x_n, y_n)]$.

```
mnist = torchvision.datasets.MNIST(
    root = 'pytorch-data/',      # where to download the dataset to
    download = True,       # if files aren't here, download them
    train = True,          # whether to import the test or the train subset
    transform = torchvision.transforms.ToTensor() #convert x_i and y_i to torch.tensor
)
```

There are several things to look out for in the code that follows. They are all part of the craft of neural network architecture, orthogonal to probabilistic modelling concerns, so we won't do more than just mention them briefly.

- The nn.Sequential module on line 4 applies a succession of functions, one after the other. Each of the functions in the list is a module, and most have their own parameters. The modules used here have been found useful for image processing; see specialist texts about image processing to find out what they do.
- Rather than trying to optimize the entire dataset, on line 27 we're taking batches of 5 images at a time and updating the neural network weights based on the batch.
- It's been found that randomized dropout, lines 9 and 13, tends to result in better results on holdout data. The dropout module has slightly different behaviours depending on whether we're training or just taking readouts. So we need to tell our model whether it should be in training mode or not, lines 29 and 47.

```
1  class MyImageClassifier(nn.Module):
2      def __init__(self):
3          super().__init__()
4          self.f = nn.Sequential(        # input.shape [B × 1 × 28 × 28]
5              nn.Conv2d(1, 32, 3, 1),     # → [B × 32 × 26 × 26]
6              nn.ReLU(),
7              nn.Conv2d(32, 64, 3, 1),    # → [B × 64 × 24 × 24]
8              nn.MaxPool2d(2),            # → [B × 64 × 12 × 12]
9              nn.Dropout2d(0.25),
10             nn.Flatten(1),              # → [B × 9216]
11             nn.Linear(9216, 128),       # → [B × 128]
12             nn.ReLU(),
13             nn.Dropout2d(0.5),
            nn.Linear(128, 10)          # → [B × 10]
        )
    # compute log likelihood for a batch of data
    def forward(self, x, y):   # x.shape [B × 1 × 28 × 28], y.shape and output.shape [B]
        return - nn.functional.cross_entropy(self.f(x), y, reduction='none')

    # compute the class probabilities for a single image
    def classify(self, x):                  # input.shape [1 × 28 × 28]
        q = self.f(torch.as_tensor(x)[None,...])[0]
        return nn.functional.softmax(q, dim=0)      # output.shape [10]

model = MyImageClassifier()
epoch = 0
mnist_batched = torch.utils.data.DataLoader(mnist, batch_size=5)

model.train(mode=True)
optimizer = optim.Adam(model.parameters())

with Interruptable() as check_interrupted:
    for batch_num, (imgs,lbls) in enumerate(mnist_batched):
        check_interrupted()
        optimizer.zero_grad()
        loglik = model(imgs, lbls)
        e = - torch.mean(loglik)
        e.backward()
        optimizer.step()

        if batch_num % 25 == 0:
            IPython.display.clear_output(wait=True)
            print(f'epoch={epoch} batch={batch_num}/{len(mnist_batched)} loglik={-e.item()}')
    epoch += 1

# Apply the trained network to classify a single image
model.train(mode=False)
img,_ = mnist[110]
model.classify(img)
```