Guidebook to Exercise 6

Prof. Dr. Ivan Habernal
Yassine Thlija

2025-11-18

Introduction

Welcome! This is your quick and hopefully gentle guidebook for some essential PyTorch concepts you'll be needing throughout exercise 6.

1 Random Tensors and Parameters

1.1 `torch.randn(())`

This creates a single random number drawn from a normal distribution (mean 0, std 1). Yes, the double parentheses look weird, but it's just PyTorch's way of saying "scalar, please."

```
x = torch.randn(())
```

Use this when you need a random starting value (like a weight).

1.2 `torch.randn((), requires_grad=True)`

Same deal as above, but now PyTorch will keep track of operations applied to this value so it can compute gradients later.

```
w = torch.randn((), requires_grad=True)
```

Perfect for learnable parameters.

2 Autograd Essentials

2.1 `loss.backward()`

This is where the magic happens. Once you compute your loss, calling `loss.backward()` tells PyTorch: "Alright, compute the gradients for everything that has `requires_grad=True`."

```
loss = (y_pred - y_true)**2
loss.backward()
```

After this, every parameter will get a `.grad` value.

2.2 `with torch.no_grad()`

Sometimes you want to update values, but you do not want PyTorch to track these updates computationally (like when adjusting weights manually). This context manager disables gradient tracking.

```
with torch.no_grad():
    w -= 0.01 * w.grad
```

You use this every time you update parameters without messing up the computation graph.

3 Building Your Own Autograd Functions

3.1 `torch.autograd.Function`

This lets you define your own custom forward and backward logic. It's more advanced, but very cool when PyTorch doesn't support some math directly.

```
class MyFunction(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input):
        # save stuff for backward
        return f(input)

    @staticmethod
    def backward(ctx, grad_output):
        # compute gradient manually
        return grad_output * f_prime
```

3.2 The `ctx` object

The `ctx` (context) is how you pass information from `forward` to `backward`.

3.2.1 `ctx.save_for_backward()`

Use this to store tensors that you'll need during backprop.

```
ctx.save_for_backward(input_tensor)
```

Then, inside `backward`, you can retrieve them:

```
(input_tensor,) = ctx.saved_tensors
```

4 Losses, Optimizers, and Parameters

4.1 `torch.nn.MSELoss`

This is PyTorch's built-in Mean Squared Error loss.

```
criterion = torch.nn.MSELoss()
loss = criterion(y_pred, y_true)
```

No need to compute the formula manually.

4.2 `torch.optim.SGD`

This is the classic stochastic gradient descent optimizer.

```
optimizer = torch.optim.SGD([w], lr=0.01)
```

It will update your parameters for you (once you ask nicely).

4.3 `torch.nn.Parameter`

A wrapper that tells PyTorch: "Hey, this tensor is a learnable parameter. Track its gradients."

```
self.weight = torch.nn.Parameter(torch.randn(()))
```

Used inside `nn.Module` classes.

5 Training Loop Essentials

5.1 `optimizer.zero_grad()`

Before computing new gradients, you must clear the old ones. Yes, PyTorch accumulates gradients by default.

```
optimizer.zero_grad()
```

5.2 `optimizer.step()`

After calling `loss.backward()`, you call `optimizer.step()` to actually update the parameters.

```
optimizer.step()
```

It's the "do the update now" button.