Pytorch Documentation

Chaitanya Kapoor

January 13, 2022

Contents

A list of important stuff in **PyTorch** that I do not want to keep using the official documentation for. I have made this using the book *"Deep Learning with PyTorch"*.

1 Tensor Manipulation

Tensors are indexed similar to multidimensional arrays. A Python list can also be passed to the constructor to create a tensor.

```
points = torch.tensor([4.0, 1.0, 5.0, 3.0, 2.0, 1.0])
```

The shape of the created tensor can also be accessed. Note that `shape` is **not** a callable function.

```
points.shape
```

Some common ways to index a tensor (analogous to lists):

```
points[1:]      # all rows after the first (implicitly all columns)
points[1:, :]   # all columns
points[1:, 0]   # first column
points[None]    # add a dimension of size 1 (like unsqueeze)
```

1.1 Tensor dimensions

3D tensors have shapes in the following order:

```
img_t = torch.randn(3, 5, 5) # shape [channels, rows, columns]
```

A 4D image tensor has the following shape:

```
batch_t = torch.randn(2, 3, 5, 5) # shape [batch, channels, rows, columns]
```

RGB dimensions are always counted third from the end, **-3**.

1.2 In-place operations

Methods which are specific to `Tensor` objects are followed by an *underscore*. This is used to indicate that the method operates *in place* by modifying the input instead of creating a new tensor.

```
a = torch.ones(3,2)
a.zero_()
```

This code block mutates `a` and returns a null tensor of shape $(3, 2)$.

1.3 Move to GPUs

A tensor can be created on the GPU by providing the following argument to the constructor:

```
points_gpu = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]], device='cuda')
```

We can also copy a tensor created on the CPU to the GPU with the help of the `to` method.

```
points_gpu = points.to(device='cuda')
```

Basic tensor operations can also be transferred to the GPU rather than performing them on the CPU.

```
points_gpu = 2 * points.to(device='cuda')
```

1.4 Serializing tensors

To avoid retraining models from scratch, we can save weight tensors with the help of the `pickle` format. If we have an output file called `ourpoints.t`, then:

```
torch.save(points, '../ourpoints.t')
```

To load the weights, we call the `load` method.

```
points = torch.load('..ourpoints.t')
```

2 Images

2.1 Loading images

This can be done with the `imageio` library. Suppose that the path to the image file is `'../xyz.jpg'`:

```
import imageio
img_arr = imageio.imread('../xyz.jpg')
```

The image is loaded as a `NumPy` object. The layout of the dimensions is $H \times W \times C$ corresponding to **height**, **width** and **channel** respectively.

2.2 Changing layout

The layout of the image tensor can be altered using the `permute` method (as expected by `Pytorch`).

```
torch_img = torch.from_numpy(img_arr) # convert to torch tensor
out = torch_img.permute(2, 0, 1) # (C X H X W)
```

An extra dimension can be added for processing batches of images by specifying the dimensions of the tensor as $N \times C \times H \times W$ where N is the batch size of the images.

3 Learning process

3.1 Visualizing

The most foolproof method to debug code is to begin by plotting it out. In the following code snippet, we plot *raw unknown values*.

```
%matplotlib inline
from matplotlib import pyplot as plt

t_p = model(t_un, *params) # argument unpacking
fig = plt.figure(dpi=600)
plt.xlabel("Temperature (°Fahrenheit)")
plt.ylabel("Temperature (°Celsius)")
plt.plot(t_u.numpy(), t_p.detach().numpy())
plt.plot(t_u.numpy(), t_c.numpy(), 'o')
```

The argument `*params` is equivalent to passing the elements of `params` as individual arguemnts to our training model.

3.2 Backpropagation

A training loop can be written with the help of gradient descent optimizers and standard loss functions that are shipped along with the `PyTorch` packages. The following code block illustrates a training loop function:

```
def training_loop(n_epochs, optimizer, params, t_u, t_c):
    for epoch in range(1, n_epochs + 1):
        t_p = model(t_u, *params)
        loss = loss_fn(t_p, t_c) # custom loss function
        optimizer.zero_grad()
```

```
        loss.backward()
        optimizer.step() # update parameters
        if epoch % 500 == 0:
            print('Epoch %d, Loss %f' % (epoch, float(loss)))
    return params
```

Note that the gradients are *zeroed out* after each loss update to ensure that the gradients are not accumulated in the leaf nodes of the computation graph that is constructed. An example of invoking the above training method using the `SGD` optimizer is shown below.

```
params = torch.tensor([1.0, 0.0], requires_grad=True)
learning_rate = 1e-2
optimizer = optim.SGD([params], lr=learning_rate)
training_loop(
    n_epochs = 5000,
    optimizer = optimizer,
    params = params,
    t_u = t_un,
    t_c = t_c)
```

3.3 Splitting datasets

Shuffling a dataset requires us to use a random permutation of tensor indices. This is done with the help of the `randperm` function.

```
n_samples = t_u.shape[0]
n_val = int(0.2 * n_samples)    # perform an 80:20 split
shuffled_indices = torch.randperm(n_samples)
train_indices = shuffled_indices[:-n_val]
val_indices = shuffled_indices[-n_val:]
```

4 Neural networks

4.1 Linear models

The `nn.Linear` constructor takes in **three** arguments:

1. Number of input features
2. Number of output features
3. Bias (True or False)

```python
import torch.nn as nn

linear_model = nn.Linear(1, 1) # bias is 'True' by default
linear_model(t_un_val)
```

The linear module so constructed can now be called like a regular function, by passing in the input matrix as an argument. To determine the parameters of the linear model, we can call the `parameters` function on `nn.Module` to return a list of parameters. For instance, in the above linear model, the following can be done to obtain the parameters:

```python
list(linear_model.parameters())
```

4.2 Sequential models

Multiple `nn` modules can be concatenated with the help of the `nn.Sequential` container. If suppose we were to construct a simple neural network containg 2 *hidden* layers with a `Tanh` activation, the following code snippet would construct the required model:

```python
seq_model = nn.Sequential(
            nn.Linear(1, 13),
            nn.Tanh(),
            nn.Linear(13, 1))
```

In essence, the model constructed above has 1 input feature which fans out to 13 hidden features which is then passed through a `Tanh` activation. This combines linearly to give a single output feature.

5 Basic vision

5.1 Datasets

5.1.1 Download

We make use of the `datasets` module to download standard datasets. For example, we would download the **CIFAR-10** dataset in the following manner:

```python
from torchvision import datasets
data_path = '/path/to/download' # download to this directory path
cifar10 = datasets.CIFAR10(data_path, train=True,
```

```
                                  download=True) # training data
cifar10_val = datasets.CIFAR10(data_path, train=False,
                                   download=True) # validation data
```

5.1.2 Dataset class

The length of the dataset object created above can be determined using `len`, which is a built-in Python function.

```
len(cifar10)
```

Similarly, we can use standard subscripting as done for tuples and lists to access individual elements of the dataset.

```
img, label = cifar10[99] # 99th image of the dataset
```

5.1.3 Transform

In most cases, we would need to convert images which are in the form of PIL images or NumPy arrays to tensors.

```
from torchvision import transforms
to_tensor = transforms.ToTensor()
img_t = to_tensor(img)
```

`img_t` is laid out in the format $C \times H \times W$. For the sake of brevity, this can also be passed directly as an argument to `dataset.CIFAR10`:

```
tensor_cifar10 = datasets.CIFAR10(data_path, train=True, download=False,
                                     transform=transforms.ToTensor())
```

5.1.4 Normalize

Suppose that we wanted to normalize the CIFAR10 dataset. To do so, we need to compute the mean as well as standard deviation per channel. We begin by stacking all image tensors along an extra dimension.

```
imgs = torch.stack([img_t for img_t, _ in tensor_cifar10], dim=3)
```

And now, we are in a position to compute the desired values.

```
imgs.view(3, -1).mean(dim=1) # 3X1024 vector
imgs.view(3, -1).std(dim=1) # standard deviation
```

The argument for normalization can then be added to `datasets.CIFAR10` with the help of the `transforms.Normalize` function by giving the mean and standard deviation as arguments to the latter.

5.2 Dataloaders

The DataLoader class helps in shuffling the data and organizing it in mini-batches. A sample training loop for it looks like the code snippet below.

```
import torch
import torch.nn as nn

train_loader = torch.utils.data.DataLoader(cifar2, batch_size=64,
                                           shuffle=True)

model = nn.Sequential(
            nn.Linear(3072, 512),
            nn.Tanh(),
            nn.Linear(512, 2),
            nn.LogSoftmax(dim=1))

learning_rate = 1e-2

optimizer = optim.SGD(model.parameters(), lr=learning_rate)
loss_fn = nn.NLLLoss()
n_epochs = 100

for epoch in range(n_epochs):
    for imgs, labels in train_loader:
        batch_size = imgs.shape[0]
        outputs = model(imgs.view(batch_size, -1))
        loss = loss_fn(outputs, labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```