PYTORCH 101

Ritwick Chaudhry

Credits:
11-785
Soumith Chintala’s PyTorch tutorial
WHAT IS PYTORCH

- PyTorch is a scientific computing package, just like Numpy. What makes it different?

- It’s optimized for leveraging the power of GPUs (Graphics Processing Unit)

- Also, it’s deeply embedded in Python, which makes it extremely easy to use
THE POWER OF PYTORCH

- GPU support for parallel computation
- Some basic neural layers to combine in your models
- Enforce a general way to code your models
- And most importantly, automatic backpropagation
TENSORS

- Tensors are very similar to `numpy.ndarray`, with the extra support of performing operations on those on GPUs.

- Thus we have to tell PyTorch where we want to place these tensors and be careful when performing operations.

- Let’s have a look at Tensors in action!
AUTOGRAD! - CONVENTIONAL PIPELINE

- Initialize parameters

- Repeat until convergence:
  - Compute Loss
  - Compute gradients of the Loss function w.r.t parameter
  - Update parameters
AUTOGRAD! - CONVENTIONAL PIPELINE

- Initialize parameters
- Repeat until convergence:
  - Compute Loss
  - Compute gradients of the Loss function w.r.t parameter
- Update parameters

The autograd package provides automatic differentiation for all operations on Tensors.
AUTOGRAD!

- It is a define-by-run framework, which means that your backprop is defined by how your code is run, and that every single iteration can be different.

- `torch.Tensor` is the central class of the package. If you set its attribute `.requires_grad = True`, it starts to track all operations on it. When you finish your computation you can call `.backward()` and have all the gradients computed automatically. The gradient for this tensor will be accumulated into `.grad` attribute.

- To stop a tensor from tracking history, you can call `.detach()` to detach it from the computation history, and to prevent future computation from being tracked.

- To prevent tracking history (and using memory), you can also wrap the code block in `with torch.no_grad():`
A Neural Network, as we know, is just a composition of operations, to yield highly complex functions.

`torch.nn` provides a very easy way to implement Neural Networks by stacking different basic layers!

It relies on `torch.autograd` to calculate the gradients for each of the model parameters, and thus we don’t need to worry about implementing the backpropagation.

Let’s implement a very simple NN now!
SAVING AND LOADING MODELS

Saving

```python
In [24]:
print(net.state_dict().keys())
print(optimizer.state_dict().keys())
ckpt = {
    'params': net.state_dict(),
    'optim': optimizer.state_dict()
}
torch.save(ckpt, 'ckpt.pth')

dict_keys(['state', 'param_groups'])
```

Loading

```python
In [27]:
ckpt = torch.load('ckpt.pth')
net.load_state_dict(ckpt['params'], strict=True)
optimizer.load_state_dict(ckpt['optim'])
```
import torch
from torch.utils import data

class Dataset(data.Dataset):
    'Characterizes a dataset for PyTorch'
    def __init__(self, list_IDs, labels):
        'Initialization'
        self.labels = labels
        self.list_IDs = list_IDs

    def __len__(self):
        'Denotes the total number of samples'
        return len(self.list_IDs)

    def __getitem__(self, index):
        'Generates one sample of data'
        # Select sample
        ID = self.list_IDs[index]

        # Load data and get label
        X = torch.load('data/' + ID + '.pt')
        y = self.labels[ID]

        return X, y
WORKING WITH DATA LOADERS

Dataloader

```
for x, y in dataloader:
    output = model(x)
    loss = criterion(output, y)
```

```
CLASS torch.utils.data.DataLoader(dataset, batch_size=1, shuffle=False, sampler=None,
batch_sampler=None, num_workers=0, collate_fn=None, pin_memory=False,
drop_last=False, timeout=0, worker_init_fn=None, multiprocessing_context=None)
```
TORCHVISION TRANSFORMS

Pre-processing

```python
torchvision.transforms.Normalize(mean, std, inplace=False)
```

```python
torchvision.transforms.ToTensor
```

```python
torchvision.transforms.RandomResizedCrop(size, scale=(0.08, 1.0), ratio=(0.75, 1.333333333333333), interpolation=2)
```

```python
torchvision.transforms.RandomRotation(degrees, resample=False, expand=False, center=None, fill=0)
```

```python
torchvision.transforms.RandomHorizontalFlip(p=0.5)
```

```python
torchvision.transforms.RandomGrayscale(p=0.1)
```

Augmentation

```python
>>> transforms.Compose([
>>>     transforms.CenterCrop(10),
>>>     transforms.ToTensor(),
>>> ])
```

Composing them
CRASH COURSE INTO TENSORBOARD

```python
from torch.utils.tensorboard import SummaryWriter

# default 'log_dir' is "runs" - we'll be more specific here
writer = SummaryWriter('runs/fashion_mnist_experiment_1')

# ...log the running loss
writer.add_scalar('training loss',
                  running_loss / 1000,
                  epoch * len(trainloader) + i)

tensorboard --logdir=runs
```
CRASH COURSE INTO TENSORBOARD

# write to tensorboard
writer.add_image('four_fashion_mnist_images', img_grid)
SOME COMMON ERRORS!

- Size mismatch. (Try checking tensor.size())
- \( \ast \) is element-wise product.
- Ensure that the tensors are on the same devices!

```python
x = 2* torch.ones(2,2)
y = 3* torch.ones(2,2)
print(x * y)
print(x.matmul(y))
```

```
tensor([[ 6.,  6.],
         [ 6.,  6.]])
tensor([[ 12.,  12.],
         [ 12.,  12.]])
```
SOME COMMON ERRORS!

- `view()` v/s `.transpose()`
SOME COMMON ERRORS!

- OOM error!

```python
net = nn.Sequential(nn.Linear(2048, 2048), nn.ReLU(),
                    nn.Linear(2048, 2048), nn.ReLU(),
                    nn.Linear(2048, 2048), nn.ReLU(),
                    nn.Linear(2048, 2048), nn.ReLU(),
                    nn.Linear(2048, 2048), nn.ReLU(),
                    nn.Linear(2048, 120))
x = torch.ones(256, 2048)
y = torch.zeros(256).long()
net.cuda()
x.cuda()
crit=nn.CrossEntropyLoss()
out = net(x)
loss = crit(out, y)
loss.backward()
```
SOME COMMON ERRORS!

- Any guesses?

```python
net = nn.Linear(4,2)
x = torch.tensor([1,2,3,4])
y = net(x)
print(y)
```
SOME COMMON ERRORS!

- Any guesses?

```python
net = nn.Linear(4, 2)
x = torch.tensor([1, 2, 3, 4])
y = net(x)
print(y)
```

```
RuntimeError: Expected object of type torch.LongTensor but found type torch.FloatTensor
```

```python
x = x.float()
x = torch.tensor([1.0, 2.0, 3.0, 4.0])
```
SOME COMMON ERRORS!

- Anything fishy here?
SOME COMMON ERRORS!

- Anything fishy here?
SOME COMMON ERRORS!

- Identification as a parameter

```python
class MyNet(nn.Module):
    def __init__(self, n_hidden_layers):
        super(MyNet, self).__init__()
        self.n_hidden_layers = n_hidden_layers
        self.final_layer = nn.Linear(128, 10)
        self.act = nn.ReLU()
        self.hidden = []
        for i in range(n_hidden_layers):
            self.hidden.append(nn.Linear(128, 128))
        self.hidden = nn.ModuleList(self.hidden)

    def forward(self, x):
        h = x
        for i in range(self.n_hidden_layers):
            h = self.hidden[i](h)
            h = self.act(h)
        out = self.final_layer(h)
        return out
```
DEBUGGING!

Let's post on Piazza!
DEBUGGING!

10 STAGES OF DEBUGGING

You’ll learn the most this way!
DEBUGGING - TIPS!

- **Use a debugger!**
  ```python
  import pdb; pdb.set_trace()
  ```

- **Tons of online resources**, great pytorch documentation, and basically every error is somewhere on stackoverflow.

- **Use Piazza** - First check if someone else has encountered the same bug before making a new post. We will maintain an FAQ

- **Come to Office Hours!**
THAT’S ALL FOLKS!