Quick guide on PyTorch

Who is this for

For people who have adequate python experience and have worked with PyTorch and need a refresher on basics.

Table of Contents

Introduction

PyTorch is an open-source machine learning library. It provides a flexible and dynamic computational graph architecture, known as a dynamic computation graph, which allows for easy and intuitive definitions of neural networks and facilitates automatic differentiation. PyTorch is particularly favored for its user-friendly interface, ease of debugging, and efficient memory usage. It is widely used for applications such as natural language processing and computer vision, and it supports CPU and GPU computation for scalable and high-speed training of models.

TLDR: PyTorch is a library that makes building neural networks very easy and has good performance.

3 Basic terms you should know

1. Tensor

A tensor is a multi-dimensional array, or called a vector, that serves as the fundamental data structure for storing and manipulating data. Tensors are similar to NumPy's arrays, but it has added functionality to be used on GPUs etc. For the extend of this guide, and on many projects it is ok to think about tensors as just arrays or vectors.

They are also used to encode inputs and outputs of a model, and a model's parameters. They also have a feature called automatic differentiation, which is useful during backpropagation. These might be a little too technical, but if you have used PyTorch before hopefully it is making sense.

2. nn.Module

nn.Module is a base class for all neural network modules. When you are building your own model, you will need to inherit from this class in order to work with PyTorch. Inheritance encapsulates many useful features to create a neural network.

In the later section I will show how to create a model from scratch, but there are two things to note. When you create a model, you must inherit from nn.Module and call super in the constructor, and you have to define a forward pass method for the model. If you do these two steps, your model is ready to be trained.

3. nn.Parameter

nn.Parameter is a subclass of torch.Tensor, specifically used to indicate that this tensor is a model parameter. When you learn about layers in the following section, you can imagine that this is the underlying tensor to store that layer's weights.

Components to build a deep learning model

PyTorch provides a lot of ready to use layers that are common in neural networks. We will explore a subset that I find is useful to tackle general tasks.

1. nn.Identity

This layer passes the input exactly. It might seems useless, but it is used in Residual Networks.

2. nn.Linear

Applies a linear transformation to the input. If you know a little linear algebra, this is a matrix multiplication, the required arguments to the layer define the dimensions of the matrix. Optionally, you can add bias.

You will encounter this layer in the last layer of classification models, usually the number of output features will be the number of classes.

```
m = nn.Linear(20, 30, bias=False) # 20 input features, 30 output features, No bias
```

3. nn.Conv2d

Applies a 2-dimensional convolution. Very useful in computer vision and image classification tasks. A convolution layer learns to recognize and detect pattern in data. For example, a convolution layer may detect parts of an airplane, such as the tail and wings, and the little windows, when such feature is detected, it results in a high signal. The model learns to associate these signals with classes.

input channels define the 'depth' of an input. For a black and white image the input channels would be 1, [0, 1] range. For an RGB image, like CIFAR10, input channels would be 3, because your input has 3 depths to show red, green, and blue values.

output channels define how many features you want to extract from this convolution.

When you are stacking convolution layers, remember that the output channels of the previous layer becomes the input channels, so be careful to match.

```
m = nn.Conv2d(16, 32, 3, stride=2) # 16 input channels, 32 output channels, 3x3 kernel, 2 stride
n = nn.Conv2d(32, 64, 3, stride=2, padding=2) # 32 input, 64 output, and applies 2 pixel padding
```

A convolution will change the dimensions of the output. You can calculate the new dimensions with:

$$ \text{Output dimension} = [\frac{(W-K+2P)}{S}] + 1 $$

W or H, is the width and height of your input, if your image is square, you only need to use one. K defines your kernel size, P defines padding, S defines stride.

Please watch: https://www.youtube.com/watch?v=KuXjwB4LzSA

4. nn.MaxPool2d

Applies a 2D max pooling. It is a downsampling operation and reduces the spatial dimensions of the input feature maps.

Similar to a convolution, passes a window that extracts the maximum value inside that window. A great demonstration is: https://www.youtube.com/watch?v=mW3KyFZDNIQ

```
m = nn.MaxPool2d(3, stride=2) # window size 3, stride 2
```

This layer makes representation more compact and reduce computational load for following layers.

5. nn.Dropout

Dropout will randomly assign 0 to some elements of your input tensor. This is an effective technique for regularization and preventing the co-adaptation of neurons.

> Quick: Regularization is a set of techniques to reduce overfitting and improve model's ability to generalize to new data.

```
m = nn.Dropout(p=0.4) # will zero 40% of the input tensor
output = m(input_tensor) # output will have 40% zeroes.
```

6. nn.BatchNorm1d, nn.BatchNorm2d

Batch normalization standardizes the inputs, which stabilizes the learning process. It often results in faster convergence and allow for higher learning rates.

In torch, there are two useful normalization components: nn.BatchNorm1d (used for 2D or 3D input data, typically used with fully connected layers), BatchNorm2d (for 4D data, used with convolutions, images)

```
batch_norm = nn.BatchNorm2d(num_features=32) # layer expects an input with 32 channels
```

7. Activations and loss functions

There are many loss functions and activations to choose in PyTorch, the basic loss functions are L1Loss(MAE), MSELoss, and CrossEntropyLoss, depending on your goal, your choice of loss function will determine your success.

Activations are used to break linearity and they enable to learn complex non-linear relationships. There are again many to choose from: ReLU, Sigmoid, Tanh, are most popular.

The general knowledge on these components is enough to start building models. There exists so much more components from torch, like recurrent layers and transformer layers, if you are curious, the whole list is here: https://pytorch.org/docs/stable/nn.html#module-torch.nn

These were just building blocks, you will also need optimizers, data loaders, to actually build a full training suite. Right now, I am assuming you are given an interface to the data.

Simple Network

I will show a simple VGG network. This is a stable CNN model architecture that is characterized by its usage of 3x3 convolutions.

```
 import torch
import torch.nn as nn
# necessary imports

class VGG3Layer(nn.Module):
    def __init__(self, num_classes=1000):
        super(VGG3Layer, self).__init__()

        # nn.Sequential is a useful method to quickly define the pass of the model
        # if there was no nn.Sequential, you would have to call each layer one by one in forward()
        self.features = nn.Sequential(
            # Layer 1
            nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3, padding=1), # notice, we get rgb, hence 3 input_channels
            nn.BatchNorm2d(64), # notice that the input channels match the output of the previous layer
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            # Layer 2
            nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            # Layer 3
            nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )

        # the last layers are used for classification
        # you can use nn.Sequential again
        self.classifier = nn.Sequential(
            nn.Linear(256 * 4 * 4, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, num_classes),
        )

    def forward(self, x):
        # first apply the convolutions
        x = self.features(x)

        # this flattens the tensor, you can also use nn.Flatten, but here we want to show these steps explicitly
        x = x.view(x.size(0), -1)

        # finally apply the classifier
        x = self.classifier(x)
        return x

num_classes = 10  # CIFAR-10 dataset has 10 classes
model = VGG3Layer(num_classes=num_classes)

print(model)
```

Printing the model can tell you about its defined layers, in this case:

```
 VGG3Layer(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (4): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (5): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): ReLU(inplace=True)
    (7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (8): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): ReLU(inplace=True)
    (11): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Linear(in_features=4096, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=10, bias=True)
  )
)
```

How to train a model

I will provide a very simple training loop, the main goal for this document was to provide an introduction to minimum subset of building blocks and basics. A training loop is the part where your model learns, it is as important as designing a model.

There are a few key features you must know: training needs an understanding of error, this is why we use loss functions, to see how we are doing. The loss provides us with a computable gradient to improve our model. We will use the gradient to 'step' in a direction that lowers the loss.

To do these steps, in the training loop: zero the optimizer gradient, calculate the loss, compute the backpropagation from the loss, and finally step with the optimizer.

```
 import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
from torchvision.utils import make_grid
import matplotlib.pyplot as plt
# you need to import more libraries, possibly to print results

# don't worry about this for now, in a future guide we might go more in depth
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

# let's download our data, again don't worry about this part, out of scope for this document
trainset = datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
trainloader = DataLoader(trainset, batch_size=64, shuffle=True)

# if you have a gpu
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# initialize the model, VGG3Layer is the custom model we just made
num_classes = 10
model = VGG3Layer(num_classes=num_classes).to(device)

# criterion often refers to loss
# optimizer is component that updates model parameters
# you need the model, the loss, and the optimizer to train
criterion = nn.CrossEntropyLoss()
```

```python
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

# the main training loop, let's dive in
def train(model, device, train_loader, optimizer, epoch):
    model.train() # set the model to training mode
    # enabling training mode makes dropout and normalization behave differently
    # again, don't worry about it, a little out of scope

    running_loss = 0.0
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)

        # zero the gradient
        # if you don't zero the gradient, your calculations will be incorrect
        # this ensures you are not tracking unnecessary information in gradient
        optimizer.zero_grad()

        # find prediction from the model
        output = model(data)

        # calculate the loss
        loss = criterion(output, target)

        # compute the gradient, if you are familiar with backpropagation, this is the backward pass
        loss.backward()

        # update the model parameters
        optimizer.step()

        running_loss += loss.item()
        if batch_idx % 100 == 99:    # print every 100 mini-batches
            print(f'[{epoch + 1}, {batch_idx + 1:5d}] loss: {running_loss / 100:.3f}')
            running_loss = 0.0

train(model, device, trainloader, optimizer, 0)
```