Short Notes on Introduction to PyTorch

Xingzhi Zhou

2021-03-04

Summary

In this note, we tries to introduce the key codes and explanation for basic usage of pytorch.

Credits: most of codes are after pytorch official code and tutorial of David Bau.

1 Setup by Colab

The easiest way to explore PyTorch is to leverage the tool Colab. The details of usage of Colab is after [python-colab tutorial]. Based on this link, we can learn how to use python on Colab, as well as two useful modules, numpy and matplotlib.

(The running result of code is not shown in the node. We will elaborate the codes in tutorial).

2 Tensor manipulation and typical loss calculation

A torch.Tensor is a multi-dimensional matrix containing elements of a single data type. Typical data types of element are torch.float, torch.long. Tensor is the basic data class, like array to numpy. Meanwhile, almost all operations numpy contains, have similar functions in torch.

2.1 Create new tensors

This part, we will introduce the typical ways to create new tensors.

```
import torch
import numpy as np

# from list, numpy array
tensor_1 = torch.tensor([[1., -1.], [1., -1.]])
tensor_2 = torch.tensor(np.array([[1, 2, 3], [4, 5, 6]]))

print('Tensor 1\n', tensor_1)
print('Tensor 2\n', tensor_2)

# ones, eye, zeros
tensor_3 = torch.zeros([2, 4], dtype=torch.int32)
tensor_4 = torch.eye(3)
tensor_5 = torch.ones(3)

print('Tensor 3\n', tensor_3)
print('Tensor 4\n', tensor_4)
print('Tensor 5\n', tensor_5)

# from existed tensor
tensor_6 = torch.clone(tensor_5)
print("Tensor 6: a copy of tensor 5\n", tensor_6)

# Make a vector of 101 equally spaced numbers from 0 to 5.
x = torch.linspace(0, 5, 101)

# Print the first five things in x.
print('Create tensor by Linear space\n',x[:5])
```

2.2 Elementwise operations

The basic operations in torch is element-wise operations, such as plus, minus, multiplication, divide, power, etc.

```
import matplotlib.pyplot as plt

x = torch.linspace(0, 5, 101)
y1, y2 = x.sin(), x ** x.cos()
y3 = y2 - y1
y4 = y3.min()

# Print and plot some answers.
```

```
print(f'The shape of x is {x.shape}')
print(f'The shape of y1=x.sin() is {y1.shape}')
print(f'The shape of y2=x ** x.cos() is {y2.shape}')
print(f'The shape of y3=y2 - y1 is {y3.shape}')
print(f'The shape of y4=y3.min() is {y4.shape}, a zero-d
    scalar')

plt.plot(x, y1, 'red', x, y2, 'blue', x, y3, 'green')
plt.axhline(y4, color='green', linestyle='--')
plt.show()
```

2.3 Linear Algebra

Pytorch contains a large amount of functions for linear algebra. Here we just illustrate a few of them.

```
A = torch.tensor([[1,2],[3,4]])
B = torch.tensor([[1,1],[1,1]])
x = torch.tensor([1,1])
print(A.mm(B))       # matrix multiplication
print(A.mv(x))       # matrix-vector multiplication
print(x.t())         # matrix transpose
```

2.4 Loss Function

Pytorch provides various loss functions, benefiting training and research. We list two typical loss, cross entropy loss and binary version.

```
x = torch.tensor([[1.,2.],[3.,4.]]) # logits (prepared for
    softmax, usually is wx)
y = torch.tensor([0,1])             # label
print(torch.nn.CrossEntropyLoss()(x,y)) # calculate the cross
    entropy loss

x = torch.tensor([0.2, 0.7])          # value after sigmoid
```

```
y = torch.tensor([0.,1.])            # label
print(torch.nn.BCELoss()(x,y))       # calculate the binary
    cross entropy loss
```

2.5 Autograd

The most significant part of pytorch is autograd framework. If set 'x.requires_grad=True', pytorch will automatically keep tracking the computational history of the tensors derived from 'x'. Thus, pytorch is able to provide the derivatives of 'x' based on computation.

The function ‘torch.autograd.grad(output_scalar, [list of input_tensors])‘ computes ‘d(output_scalar)/d(input_tensor)‘ for each input tensor component in the list. For it to work, the input tensors and output must be part of the same ‘requires_grad=True‘ compuation.

```
import torch
from matplotlib import pyplot as plt

x = torch.linspace(0, 5, 100, requires_grad=True)
y = (x**2).cos()
dydx = torch.autograd.grad(y.sum(), [x])[0]

plt.plot(x.detach(), y.detach(), label='y')
plt.plot(x.detach(), dydx, label='dy/dx')
plt.legend()
plt.show()
```

```
x = torch.linspace(0, 5, 100, requires_grad=True)
y = (x**2).cos()
y.sum().backward()   # populates the grad attribute below.
print(x.grad)

plt.plot(x.detach(), y.detach(), label='y')
plt.plot(x.detach(), x.grad, label='dy/dx')
plt.legend()
plt.show()
```

3 Optimizer

After computation on gradient, we can leverage optimizer to conduct gradient descent.

```
import torch

#@title Run this cell to setup visualization...
# This cell defines plot_progress() which plots an
    optimization trace.

import matplotlib
from matplotlib import pyplot as plt

# This function is to facilitate the plot, just skip it.
def plot_progress(bowl, track, losses):
    # Draw the contours of the objective function, and x, and
    y
    fig, (ax1, ax2) = plt.subplots(1,2, figsize=(12, 5))
    for size in torch.linspace(0.1, 1.0, 10):
        angle = torch.linspace(0, 6.3, 100)
        circle = torch.stack([angle.sin(), angle.cos()])
        ellipse = torch.mm(torch.inverse(bowl), circle) * size
        ax1.plot(ellipse[0,:], ellipse[1,:], color='skyblue')
    track = torch.stack(track).t()
    ax1.set_title('progress of x')
    ax1.plot(track[0,:], track[1,:], marker='o')
    ax1.set_ylim(-1, 1)
    ax1.set_xlim(-1.6, 1.6)
    ax1.set_ylabel('x[1]')
    ax1.set_xlabel('x[0]')
    ax2.set_title('progress of y')
    ax2.xaxis.set_major_locator(matplotlib.ticker.MaxNLocator(
    integer=True))
    ax2.plot(range(len(losses)), losses, marker='o')
    ax2.set_ylabel('objective')
    ax2.set_xlabel('iteration')
    fig.show()

x_init = torch.randn(2)
x = x_init.clone()
x.requires_grad = True
optimizer = torch.optim.SGD([x], lr=0.1, momentum=0.5)

bowl = torch.tensor([[ 0.4410, -1.0317], [-0.2844, -0.1035]])
track, losses = [], []

for iter in range(100):
    objective = torch.mm(bowl, x[:,None]).norm()
    optimizer.zero_grad()
    objective.backward()
    optimizer.step()
    track.append(x.detach().clone())
    losses.append(objective.detach())

plot_progress(bowl, track, losses)
```

4 Modules

We will illustrate Linear module here, which is a fundamental workhorse of all neural networks. After this, we can better understand combined complex modules.

```
import torch
net = torch.nn.Linear(3, 2)
```

```python
print(net)

print(net(torch.tensor([[1.0, 0.0, 0.0]])))

x_batch = torch.tensor([
[1.0, 0. , 0. ],
[0. , 1.0, 0. ],
[0. , 0. , 1.0],
[0. , 0. , 0. ],
])

print(net(x_batch))

print('weight is', net.weight)
print('bias is', net.bias)

y_batch = net(x_batch)
loss = ((y_batch - torch.tensor([[1.0, 1.0]])) ** 2).sum(1).
    mean()
print(f'loss is {loss}')

loss.backward()
print(f'weight is {net.weight} and grad is:\n{net.weight.grad
    }\n')
print(f'bias is {net.bias} and grad is:\n{net.bias.grad}\n')
```

```python
net = torch.nn.Linear(3, 2)
log = []
for _ in range(10000):
    y_batch = net(x_batch)
    loss = ((y_batch - torch.tensor([[1.0, 1.0]])) ** 2).sum
    (1).mean()
    log.append(loss.item())
    net.zero_grad()
    loss.backward()
    with torch.no_grad():
        for p in net.parameters():
            p[...] -= 0.01 * p.grad

print(f'weight is {net.weight}\n')
print(f'bias is {net.bias}\n')

%matplotlib inline
import matplotlib.pyplot as plt
plt.ylabel('loss')
plt.xlabel('iteration')
plt.plot(log)
```

For self-defined module, please see details in official introduction on CUSTOM NN MODULES

```
class TwoLayerNet(torch.nn.Module):
def __init__(self, D_in, H, D_out):
    """
    In the constructor we instantiate two nn.Linear modules
   and assign them as
    member variables.
    """
    super(TwoLayerNet, self).__init__()
    self.linear1 = torch.nn.Linear(D_in, H)
    self.linear2 = torch.nn.Linear(H, D_out)

def forward(self, x):
    """
    In the forward function we accept a Tensor of input data
   and we must return
```

```
    a Tensor of output data. We can use Modules defined in the
     constructor as
    well as arbitrary operators on Tensors.
    """
    h_relu = self.linear1(x).clamp(min=0)
    y_pred = self.linear2(h_relu)
    return y_pred
```

5 Datasets and DataLoader

Datasets class in pytorch helps to keep dataset and get items.

DataLoader class helps to make a batch of data based on given sampler and parameters.

One example is shown as blow:

```
# The simplest way to leverage Dataset and DataLoader
import torch
a = torch.randn(10,2)                          # feature
b = torch.randn(10,1)                          # target
dataset_ = torch.utils.data.TensorDataset(a,b) # Construct the
     dataset
loader_ = torch.utils.data.DataLoader(dataset_,
batch_size = 5) # Construct loader
# Check what's inside
for input, target in loader_:
    print('input',input,'\n target',target)
    print('-'*40)
```

Further, you can download the dataset by torchvision, see details. Also, you can custom your own dataset and data loader, see details(note: the pictures used in this link may make you uncomfortable, though it is an official tutorial by pytorch)

6 Train on GPU and parallel on GPU

To train on GPU, you only need to make sure your model and data are CUDA tensor types, i.e. they are on GPU devices. "CUDA (an acronym for Compute Unified Device Architecture) is a parallel computing platform and application programming interface (API) model created by Nvidia." - from Wikipedia.

```
# check cuda availability
print('Cuda avaiblble?', torch.cuda.is_available())

# A compromise way to set cuda, by detecting cuda availability
device = torch.device('cuda' if torch.cuda.is_available else '
    cpu')
print('device is :', device)

# convert the tensor
x = torch.randn(2,2)
print('cpu tensor \n', x)
x = x.to(device)
print('After changing device\n',x)

# conver the module
layer = torch.nn.Linear(2,2)
print(list(layer.parameters()))
layer = layer.to(device)
print(list(layer.parameters()))
```

To achieve parallel training on GPU, pytorch provides a helpful wrapper.

```
#The easiest to use parallel is to wrap model.
model = nn.DataParallel(model)

# To make the program more robust, we need to check GPU amount
    .
```

```
if torch.cuda.device_count() > 1:
    print("Let's use", torch.cuda.device_count(), "GPUs!")
    model = nn.DataParallel(model)
```

However, we need to be careful about parallel wrapped model at training, saving, loading.

In training, the loss will become multiple. To deal with this, use following before back propagation.

```
loss =  torch.mean(loss)
```

In saving, we have to change the model saving to

```
torch.save(model.module.state_dict(), #FILE PATH#)
```

in loading, we have to change the model loading to

```
model.module.load_state_dict(torch.load(#FILE PATH#))
```

7 Useful Tools and Links

This section will list several useful tools when you are training models with PyTorch.

- logging
- tensorboardX
- jupyterlab / jupyter notebook
- pycharm a useful IDE to play with Python

The following are related tutorials on the websites:

- The tutorial from David Bau
- official tutorial of pytorch