

3. Basic use of PyTorch



Raspberry Pi motherboard series does not currently support PyTorch functions.

3.1. About PyTorch

3.1.1. Introduction

PyTorch is an [open source Python](#) machine learning library based on Torch for applications such as natural language processing.

3.1.2. Features

- 1), Powerful GPU-accelerated tensor calculation
- 2), Deep neural network of automatic differentiation system
- 3), Dynamic graph mechanism

3.2. Tensors in PyTorch

3.2.1. Tensors

Tensors are called Tensors in English. They are the basic computing units in PyTorch. They represent a multi-dimensional matrix just like Numpy's ndarray. The biggest difference from ndarray is that PyTorch's Tensor can run on GPU, while Numpy's ndarray can only run on CPU. Running on GPU greatly speeds up the computing speed.

3.2.2. create tensors

1), there are many ways to create tensors, calling different API interfaces can create different types of tensors,

a = torch.empty(2,2): create an uninitialized 2*2 tensor

b = torch.rand(5, 6): create a uniformly distributed initialized tensor with each element from 0-1

c = torch.zeros(5, 5, dtype=torch.long): create an initialized all-zero tensor and specify the type of each element as long

d = c.new_ones(5, 3, dtype=torch.double): create a new tensor d based on the known tensor c

d.size(): get the shape of tensor d

2), operations between tensors

The operations between tensors are actually operations between matrices. Due to the dynamic graph mechanism, mathematical calculations can be performed directly on tensors, for example,

- Add two tensors:

```
c = torch.zeros(5,3,dtype=torch.long)
d = torch.ones(5,3,dtype=torch.long)
e = c + d
print(e)
```

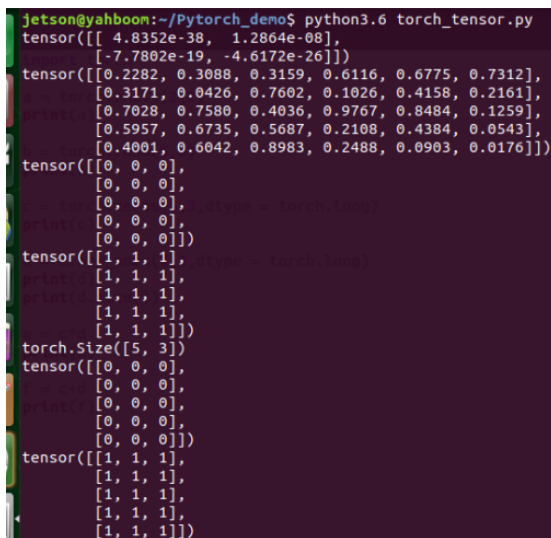
- Multiply two tensors

```
c = torch.zeros(5,3,dtype=torch.long)
d = torch.ones(5,3,dtype=torch.long)
e = c * d
print(e)
```

For this part of the code, please refer to: /home/yahboom/YBAMR-COBOT-EDU-00001/src/yahboom_navrobo_other/Pytorch/torch_tensor.py

Run the code,

```
python torch_tensor.py
```



```
jetson@yahboom:~/Pytorch_demo$ python3.6 torch_tensor.py
tensor([[ 4.8352e-38,  1.2864e-08],
        [-7.7802e-19, -4.6172e-26]])
tensor([[0.2282, 0.3088, 0.3159, 0.6116, 0.6775, 0.7312],
        [0.3171, 0.0426, 0.7602, 0.1026, 0.4158, 0.2161],
        [0.7028, 0.7580, 0.4036, 0.9767, 0.8484, 0.1259],
        [0.5957, 0.6735, 0.5687, 0.2108, 0.4384, 0.0543],
        [0.4001, 0.6042, 0.8983, 0.2488, 0.0903, 0.0176]])
tensor([[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]])
tensor([[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]])
tensor([[1, 1, 1],
        [1, 1, 1],
        [1, 1, 1],
        [1, 1, 1],
        [1, 1, 1]])
torch.Size([5, 3])
tensor([[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]])
tensor([[1, 1, 1],
        [1, 1, 1],
        [1, 1, 1],
        [1, 1, 1],
        [1, 1, 1]])
```

3.3, torchvision package introduction

3.3.1, torchvision is a library in Pytorch specifically used to process images, which contains four major categories:

1), torchvision.datasets: load datasets, Pytorch has many datasets such as CIFAR, MNIST, etc., you can use this class to load datasets, the usage is as follows:

```
cifar_train_data = torchvision.datasets.CIFAR10(root='./data', train=True,
                                                  download=False, transform=transform)
```

2), torchvision.models: load trained models, including the following VGG, ResNet, etc., the usage is as follows:

```
import torchvision.models as models
resnet18 = models.resnet18()
```

3), torchvision.transforms: Class for image conversion operations, usage is as follows:

```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
```

4), torchvision.utils: Arrange the images into a grid shape, usage is as follows:

```
torchvision.utils.make_grid(tensor, nrow=8, padding=2, normalize=False,
                             range=None, scale_each=False, pad_value=0)
```

For more information about the use of the torchvision package, please refer to the official website documentation: <https://pytorch.org/vision/0.8/datasets.html>

3.4, Convolutional Neural Network

3.4.1, Neural Network

1), Difference between Neural Network and Machine Learning

Neural Network and Machine Learning are both used for classification tasks. The difference is that Neural Network is more efficient than Machine Learning, the data is simpler, and fewer parameters are required to perform tasks. The following points are explained:

- Efficiency: The efficiency of neural network is reflected in the extraction of features. It is different from the features of machine learning. It can be trained and self-"corrected". We only need to input data into it, and it will continuously update the features.
- Data simplicity: In the process of machine learning, we need to process the data before inputting it, such as normalization, format conversion, etc., but in neural network, too much processing is not required.
- Fewer parameters for performing tasks: In machine learning, we need to adjust the penalty factor, slack variable, etc. to adjust to the most suitable effect, but for neural network, only a weight w and bias term b are given. These two values will be constantly corrected during the training process and adjusted to the optimal value to minimize the error of the model.

3.4.2, Convolutional Neural Network

1), Convolution Kernel

Convolution kernel can be understood as feature extractor, filter (digital signal processing), etc. Neural network has three layers (input layer, hidden layer, output layer), neurons in each layer can share convolution kernel, so it is very convenient to process high-order data. We only need to design the size, number and sliding step of convolution kernel to let it train itself.

2), Three basic layers of convolutional neural network:

- Convolution layer

Perform convolution operation, inner product operation of two matrices of convolution kernel size, multiply and add numbers in the same position. Convolution layer close to the input layer sets a small number of convolution kernels, and the more convolution kernels are set in the later layer, the more convolution kernels are set.

- Pooling layer

Compress images and parameters by downsampling, but will not damage the quality of the image. There are two pooling methods, MaxPooling (that is, taking the largest value in the sliding window) and AveragePooling (taking the average of all values in the sliding window).

- Flatten layer & Fully Connected layer

This layer is mainly a stacking layer. After the pooling layer, the image is compressed and then enters the Flatten layer; the output of the Flatten layer is placed in the Fully Connected layer and classified using softmax.

3.5. Build LetNet neural network and train data set

3.5.1. Preparation before implementation

1) Environment

ROSMaster-jetson development board series are all installed with the development environment of this project, including:

- python 3.6+
- torch 1.8.0
- torchvision 0.9.0

2) Dataset

CIFAR-10, 50,000 training images of 32*32 size and 10,000 test images

Note: The data set is saved in the ~/Pytorch_demo/data/cifar-10-batches-py directory,

文件名	文件用途
batches.meta. bet	文件存储了每个类别的英文名称。可以用记事本或其他文本文件阅读器打开浏览查看
data batch 1.bin 、 data batch 2.bin 、 data batch 5.bin	这5个文件是CIFAR-10数据集中的训练数据。每个文件以二进制格式存储了10000张32 × 32的彩色图像和这些图像对应的类别标签。一共50000张训练图像
test batch.bin	这个文件存储的是测试图像和测试图像的标签。一共10000张
readme.html	数据集介绍文件

3.5.2. Implementation process

1) Import related modules

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms
import torch.optim as optim
```

2) load the data set

```
cifar_train_data = torchvision.datasets.CIFAR10(root='./data', train=True,
                                                download=False, transform=transform)
cifar_test_data = torchvision.datasets.CIFAR10(root='./data', train=False,
                                                transform=transform)
```

3) encapsulating the data set

```
train_data_loader = torch.utils.data.DataLoader(cifar_train_data, batch_size=32,
                                                shuffle=True)
test_data_loader = torch.utils.data.DataLoader(cifar_test_data, batch_size=32,
                                                shuffle=True)
```

4) Build a convolutional neural network

```
class LeNet(nn.Module):
    #Define the operators required by the network, such as convolution, fully
    connected operators, etc.
    def __init__(self):
        super(LeNet, self).__init__()
        #Conv2d parameter meaning: number of input channels, number of output
        channels, kernel size
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16*5*5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
        self.pool = nn.MaxPool2d(2, 2)
    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool(x)
        x = F.relu(self.conv2(x))
        x = self.pool(x)
        x = x.view(-1, 16*5*5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

5) Configure the loss function and optimizer for training

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.005, momentum=0.9)
```

6) Start training and testing

3.5.3 Run the program

1) Reference code path

```
/home/yahboom/YBAMR-COBOT-EDU-00001/src/yahboom_navrobo_other/Pytorch/pytorch_demo.py
```

2) Run the program

```
cd /home/yahboom/YBAMR-COBOT-EDU-00001/src/yahboom_navrobo_other/Pytorch
python pytorch_demo.py
```

```
jetson@yahboom:~/Pytorch_demo$ python3.6 pytorch_demo.py
Dataset CIFAR10
Number of datapoints: 50000
Root location: ./data
Split: Train
StandardTransform
Transform: Compose(
  ToTensor()
  Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5))
)
Dataset CIFAR10
Number of datapoints: 10000
Root location: ./data
Split: Test
StandardTransform
Transform: Compose(
  ToTensor()
  Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5))
)
50000
10000
开始训练...
[Epoch 1, Batch 100] loss: 2.30272
[Epoch 1, Batch 200] loss: 2.29363
[Epoch 1, Batch 300] loss: 2.21469
[Epoch 1, Batch 400] loss: 2.04387
[Epoch 1, Batch 500] loss: 1.95162
[Epoch 1, Batch 600] loss: 1.85417
[Epoch 1, Batch 700] loss: 1.78166
[Epoch 1, Batch 800] loss: 1.73354
[Epoch 1, Batch 900] loss: 1.67480
[Epoch 1, Batch 1000] loss: 1.64514
[Epoch 1, Batch 1100] loss: 1.65078
[Epoch 1, Batch 1200] loss: 1.61990
[Epoch 1, Batch 1300] loss: 1.60939
[Epoch 1, Batch 1400] loss: 1.55830
[Epoch 1, Batch 1500] loss: 1.52808
[Epoch 2, Batch 100] loss: 1.48611
[Epoch 2, Batch 200] loss: 1.47165
[Epoch 2, Batch 300] loss: 1.47499
[Epoch 2, Batch 400] loss: 1.41587
[Epoch 2, Batch 500] loss: 1.44796
[Epoch 2, Batch 600] loss: 1.43487
[Epoch 2, Batch 700] loss: 1.41381
[Epoch 2, Batch 800] loss: 1.40199
[Epoch 2, Batch 900] loss: 1.42502
[Epoch 2, Batch 1000] loss: 1.37514
[Epoch 2, Batch 1100] loss: 1.37851
[Epoch 2, Batch 1200] loss: 1.39184
[Epoch 2, Batch 1300] loss: 1.35155
[Epoch 2, Batch 1400] loss: 1.34022
[Epoch 2, Batch 1500] loss: 1.35495
训练完成！
开始测试...
10000张测试图的准确率为: 51 %
```

We only trained twice here. You can modify the epoch value to modify the number of training times. The more training times, the higher the accuracy.