# Ch1.【Pytorch】2024 Pytorch基础入门教程（完整详细版）

置顶

北村南 已于 2024-07-26 00:15:37 修改

原文链接：https://blog.csdn.net/ccaoshangfei/article/details/126074300

————————————————

目录

一、Pytorch

1.1 简介

1.2 安装

二、Tensor

2.1 Tensor创建

2.1.1 torch.tensor() && torch.tensor([])

2.1.2 torch.randn && torch.randperm

2.1.3 torch.range(begin，end，step)

2.1.4 指定numpy

2.2 Tensor运算

2.2.1 A.add() && A.add\_()

2.2.2 torch.stack

三、CUDA

3.1 使用GPU

四、其他技巧

4.1 自动微分

4.1.1 backward求导

4.1.2 autograd.grad求导

4.1.3 求最小值

4.2 Pytorch层次结构

五、数据

5.1 Dataset and DataLoader

5.2 数据读取与预处理

5.5 Pytorch工具

六、torch.nn

## 一、Pytorch

### 1.1 简介

Pytorch是torch的python版本，是由Facebook开源的神经网络框架，专门针对 GPU 加速的深度神经网络（DNN）编程。Torch 是一个经典的对多维矩阵数据进行操作的张量（tensor ）库，在机器学习和其他数学密集型应用有广泛应用。与Tensorflow的静态计算图不同，pytorch的计算图是动态的，可以根据计算需要实时改变计算图。但由于Torch语言采用 Lua，导致在国内一直很小众，并逐渐被支持 Python 的 Tensorflow 抢走用户。作为经典机器学习库 Torch 的端口，PyTorch 为 Python 语言使用者提供了舒适的写代码选择。

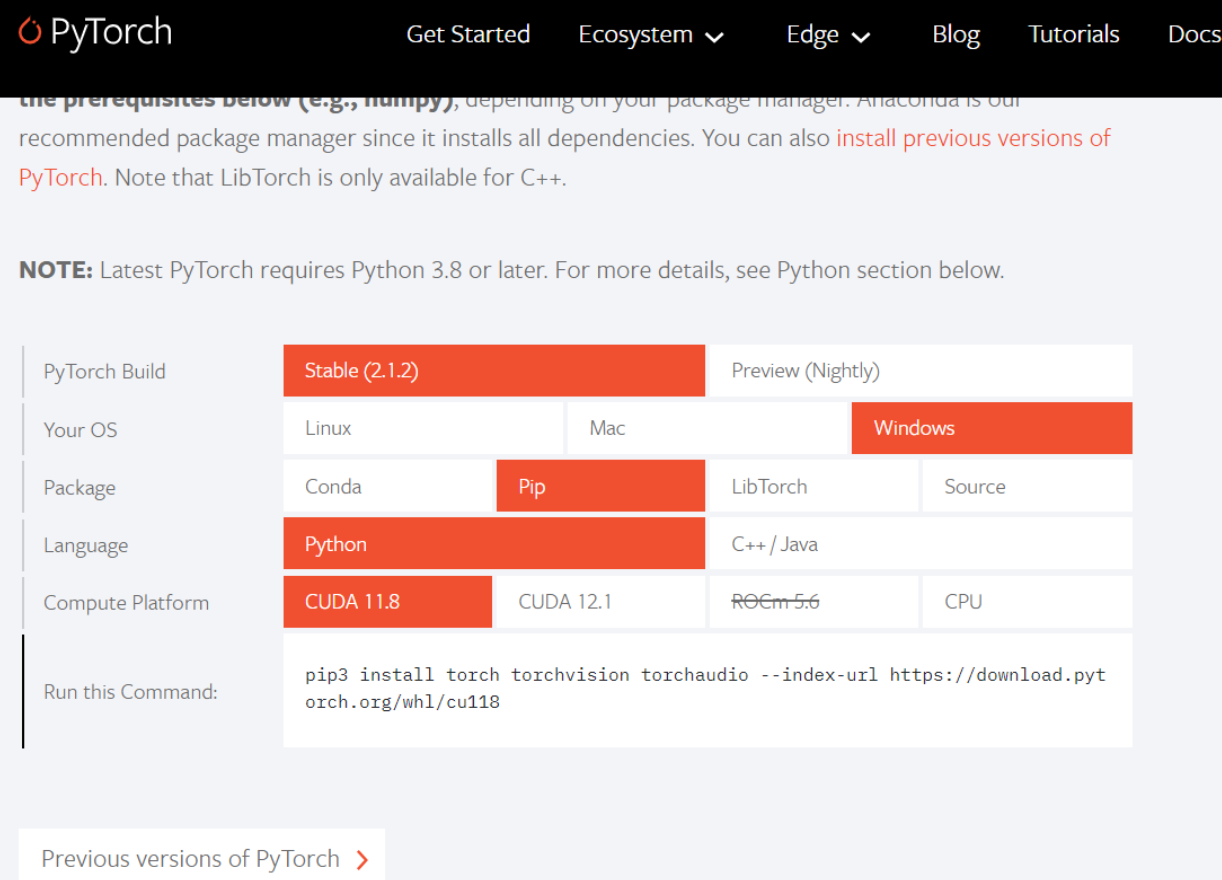
至于为什么推荐使用Pytorch，我想最主要的原因就是它非常的简洁，非常符合Python的风格。

### **1.2 安装**

首先确保你已经安装了GPU环境，即Anaconda、CUDA和CUDNN

随后进入[Pytorch官网](https://so.csdn.net/so/search?q=Pytorch%E5%AE%98%E7%BD%91&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/ccaoshangfei/article/details/_blank)[PyTorch](https://pytorch.org/" \o "PyTorch)

官网会自动显示符合你电脑配置的Pytorch版本，复制指令到conda环境中运行即可



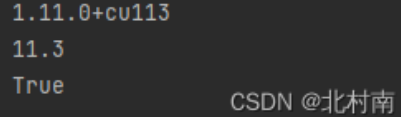
测试是否安装成功

import torch

print(torch.\_\_version\_\_) # pytorch版本

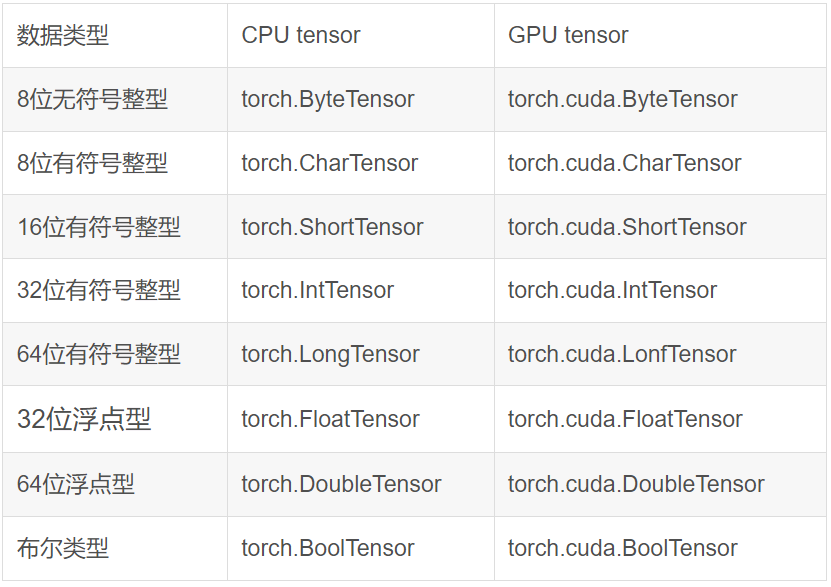
print(torch.version.cuda) # cuda版本

print(torch.cuda.is\_available()) # 查看cuda是否可用

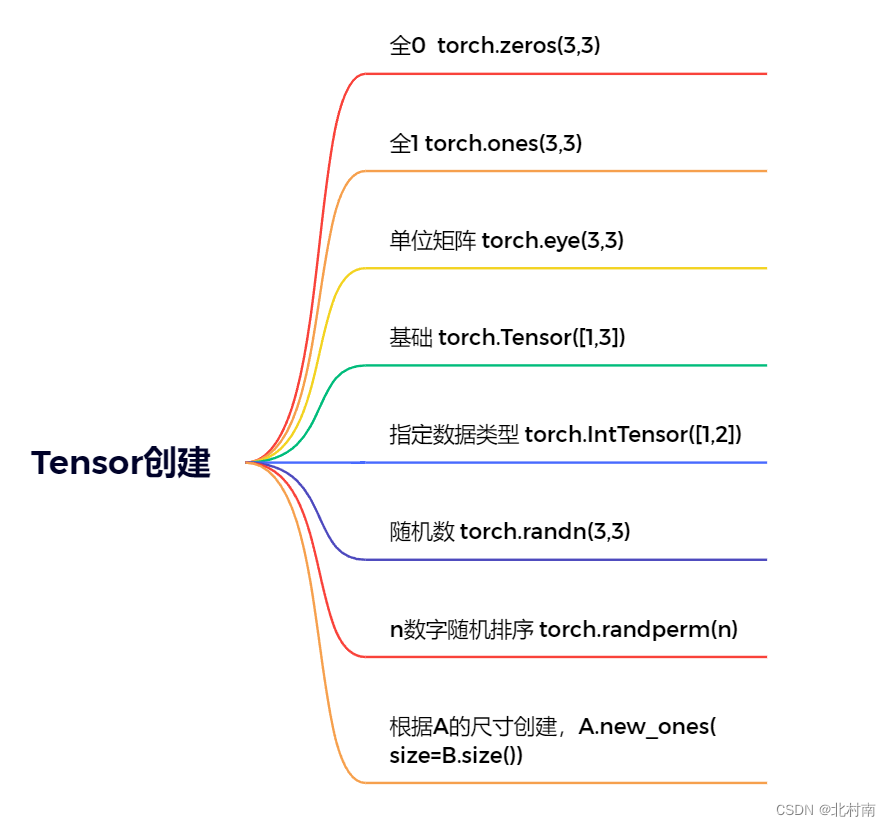


## **二、Tensor**

Tensor张量是Pytorch里最基本的数据结构。直观上来讲，它是一个多维矩阵，支持GPU[加速](https://so.csdn.net/so/search?q=%E5%8A%A0%E9%80%9F&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/ccaoshangfei/article/details/_blank)，其基本数据类型如下

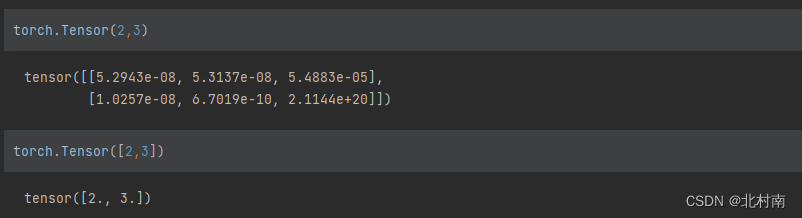


### **2.1 Tensor创建**



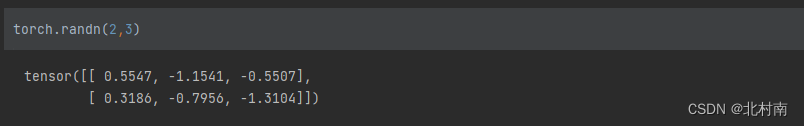
#### **2.1.1 torch.tensor() && torch.tensor([])**

二者的主要区别在于创建的对象的size和value不同

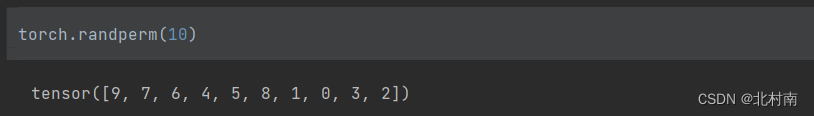


#### **2.1.2 torch.randn && torch.randperm**

生成的数据类型为浮点型，与numpy.randn生成随机数的方法类似，生成的浮点数的取值满足均值为0，方差为1的正态分布

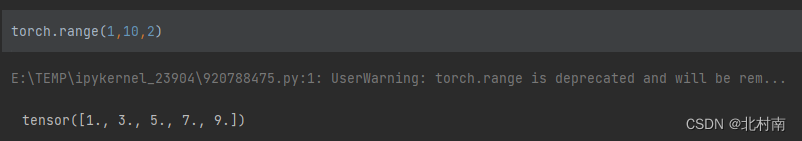


torch.randpern(n)为创建一个n个整数，随机排列的Tensor



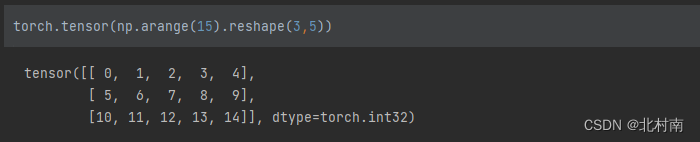
#### **2.1.3 torch.range(begin，end，step)**

生成一个一维的Tensor，三个参数分别的起始位置，终止位置和步长



#### 2.1.4 指定numpy

很多时候我们需要创建指定的Tensor，而numpy就是一个很好的方式



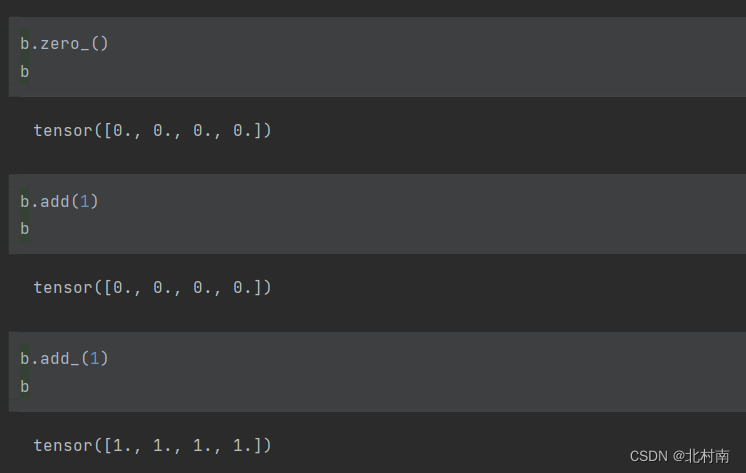
### **2.2 Tensor运算**





#### **2.2.1 A.add() && A.add\_()**

所有的带\_符号的函数都会对原数据进行修改



#### **2.2.2 torch.stack**

stack为拼接函数，函数的第一个参数为需要拼接的Tensor，第二个参数为细分到哪个维度

A=torch.IntTensor([[1,2,3],[4,5,6]])

B=torch.IntTensor([[7,8,9],[10,11,12]])

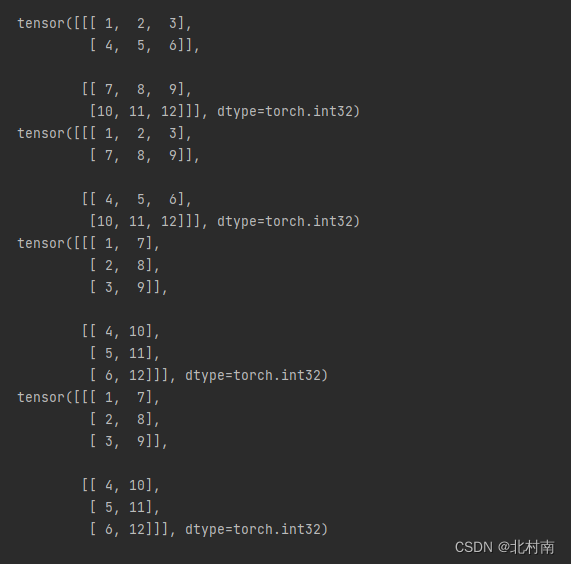
C1=torch.stack((A,B),dim=0) # or C1=torch.stack((A,B))

C2=torch.stack((A,B),dim=1)

C3=torch.stack((A,B),dim=2)

C4=torch.stack((A,B),dim=-1)

print(C1,C2,C3,C4)



dim=0，C1 = [ A,B ]

dim=1，C2 = [ [ A[0],B[0] ] , [ A[1],B[1] ] ]

dim=2，C3 = [ [ [ A[0][0],B[0][0] ] , [ A[0][1],B[0][1] ] , [ A[0][2],B[0][2] ] ],

[ [ A[1][0],B[1][0] ] , [ A[1][1],B[1][1] ] , [ A[1][2],B[1][2] ] ] ]

dim=-1，C4 = C3

## **三、CUDA**

CUDA是一种操作GPU的软件架构，Pytorch配合GPU环境这样模型的训练速度会非常的快

### **3.1 使用GPU**

import torch

# 测试GPU环境是否可使用

print(torch.\_\_version\_\_) # pytorch版本

print(torch.version.cuda) # cuda版本

print(torch.cuda.is\_available()) # 查看cuda是否可用

#使用GPU or CPU

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

# 判断某个对象是在什么环境中运行的

a.device

# 将对象的环境设置为device环境

A = A.to(device)

# 将对象环境设置为COU

A.cpu().device

# 若一个没有环境的对象与另外一个有环境a对象进行交流,则环境全变成环境a

a+b.to(device)

# cuda环境下tensor不能直接转化为numpy类型,必须要先转化到cpu环境中

a.cpu().numpy()

# 创建CUDA型的tensor

torch.tensor([1,2],device)

## **四、其他技巧**

### **4.1 自动微分**

神经网络依赖反向传播求梯度来更新网络的参数，求梯度是个非常复杂的过程，在Pytorch中，提供了两种求梯度的方式，一个是backward，将求得的结果保存在自变量的grad属性中，另外一种方式是torch.autograd.grad

#### 4.1.1 backward求导

使用backward进行求导。这里主要介绍了求导的两种对象，标量Tensor和非标量Tensor的求导。两者的主要区别是非标量Tensor求导的主要区别是加了一个gradient的Tensor，其尺寸与自变量X的尺寸一致。在求完导后，需要与gradient进行点积，所以只是一般的求导的话，设置的参数全部为1。最后还有一种使用标量的求导方式解决非标量求导，了解了解就好了。

import numpy as np

import torch

# 标量Tensor求导

# 求 f(x) = a\*x\*\*2 + b\*x + c 的导数

x = torch.tensor(-2.0, requires\_grad=True)

a = torch.tensor(1.0)

b = torch.tensor(2.0)

c = torch.tensor(3.0)

y = a\*torch.pow(x,2)+b\*x+c

y.backward() # backward求得的梯度会存储在自变量x的grad属性中

dy\_dx =x.grad

dy\_dx

# 非标量Tensor求导

# 求 f(x) = a\*x\*\*2 + b\*x + c 的导数

x = torch.tensor([[-2.0,-1.0],[0.0,1.0]], requires\_grad=True)

a = torch.tensor(1.0)

b = torch.tensor(2.0)

c = torch.tensor(3.0)

gradient=torch.tensor([[1.0,1.0],[1.0,1.0]])

y = a\*torch.pow(x,2)+b\*x+c

y.backward(gradient=gradient)

dy\_dx =x.grad

dy\_dx

# 使用标量求导方式解决非标量求导

# 求 f(x) = a\*x\*\*2 + b\*x + c 的导数

x = torch.tensor([[-2.0,-1.0],[0.0,1.0]], requires\_grad=True)

a = torch.tensor(1.0)

b = torch.tensor(2.0)

c = torch.tensor(3.0)

gradient=torch.tensor([[1.0,1.0],[1.0,1.0]])

y = a\*torch.pow(x,2)+b\*x+c

z=torch.sum(y\*gradient)

z.backward()

dy\_dx=x.grad

dy\_dx

#### **4.1.2 autograd.grad求导**

import torch

#单个自变量求导

# 求 f(x) = a\*x\*\*4 + b\*x + c 的导数

x = torch.tensor(1.0, requires\_grad=True)

a = torch.tensor(1.0)

b = torch.tensor(2.0)

c = torch.tensor(3.0)

y = a \* torch.pow(x, 4) + b \* x + c

#create\_graph设置为True,允许创建更高阶级的导数

#求一阶导

dy\_dx = torch.autograd.grad(y, x, create\_graph=True)[0]

#求二阶导

dy2\_dx2 = torch.autograd.grad(dy\_dx, x, create\_graph=True)[0]

#求三阶导

dy3\_dx3 = torch.autograd.grad(dy2\_dx2, x)[0]

print(dy\_dx.data, dy2\_dx2.data, dy3\_dx3)

# 多个自变量求偏导

x1 = torch.tensor(1.0, requires\_grad=True)

x2 = torch.tensor(2.0, requires\_grad=True)

y1 = x1 \* x2

y2 = x1 + x2

#只有一个因变量,正常求偏导

dy1\_dx1, dy1\_dx2 = torch.autograd.grad(outputs=y1, inputs=[x1, x2], retain\_graph=True)

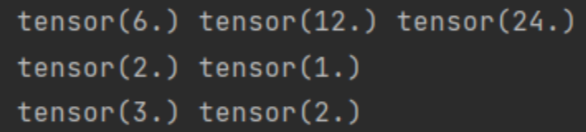
print(dy1\_dx1, dy1\_dx2)

# 若有多个因变量，则对于每个因变量,会将求偏导的结果加起来

dy1\_dx, dy2\_dx = torch.autograd.grad(outputs=[y1, y2], inputs=[x1, x2])

dy1\_dx, dy2\_dx

print(dy1\_dx, dy2\_dx)



#### **4.1.3 求最小值**

使用自动微分机制配套使用SGD随机梯度下降来求最小值

#例2-1-3 利用自动微分和优化器求最小值

import numpy as np

import torch

# f(x) = a\*x\*\*2 + b\*x + c的最小值

x = torch.tensor(0.0, requires\_grad=True)  # x需要被求导

a = torch.tensor(1.0)

b = torch.tensor(-2.0)

c = torch.tensor(1.0)

optimizer = torch.optim.SGD(params=[x], lr=0.01)  #SGD为随机梯度下降

print(optimizer)

def f(x):

    result = a \* torch.pow(x, 2) + b \* x + c

    return (result)

for i in range(500):

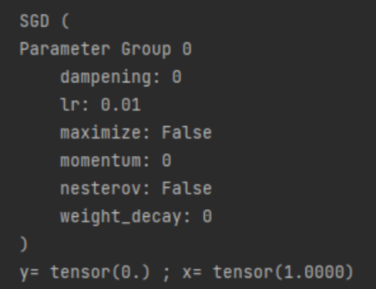
    optimizer.zero\_grad()  #将模型的参数初始化为0

    y = f(x)

    y.backward()  #反向传播计算梯度

    optimizer.step()  #更新所有的参数

print("y=", y.data, ";", "x=", x.data)



### **4.2 Pytorch层次结构**

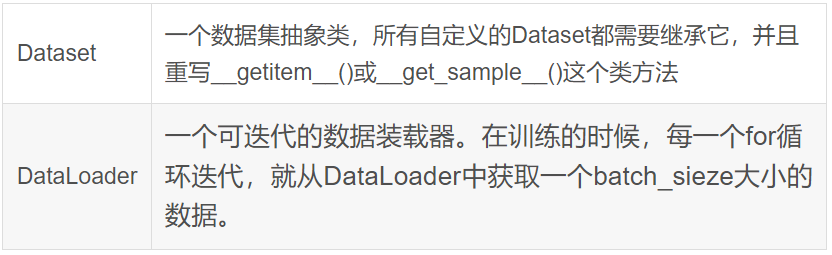
Pytorch中一共有5个不同的层次结构，分别为硬件层、内核层、低阶API、中阶API和高阶API（torchkeras）



## **五、数据**

Pytorch主要通过Dataset和DataLoader进行构建数据管道

### **5.1 Dataset and DataLoader**



### **5.2 数据读取与预处理**

DataLoader的参数如下

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,

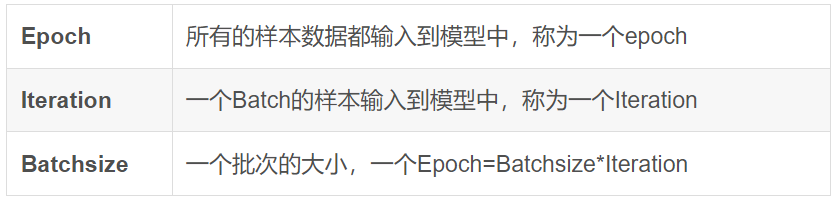
 multiprocessing\_context=None,

)

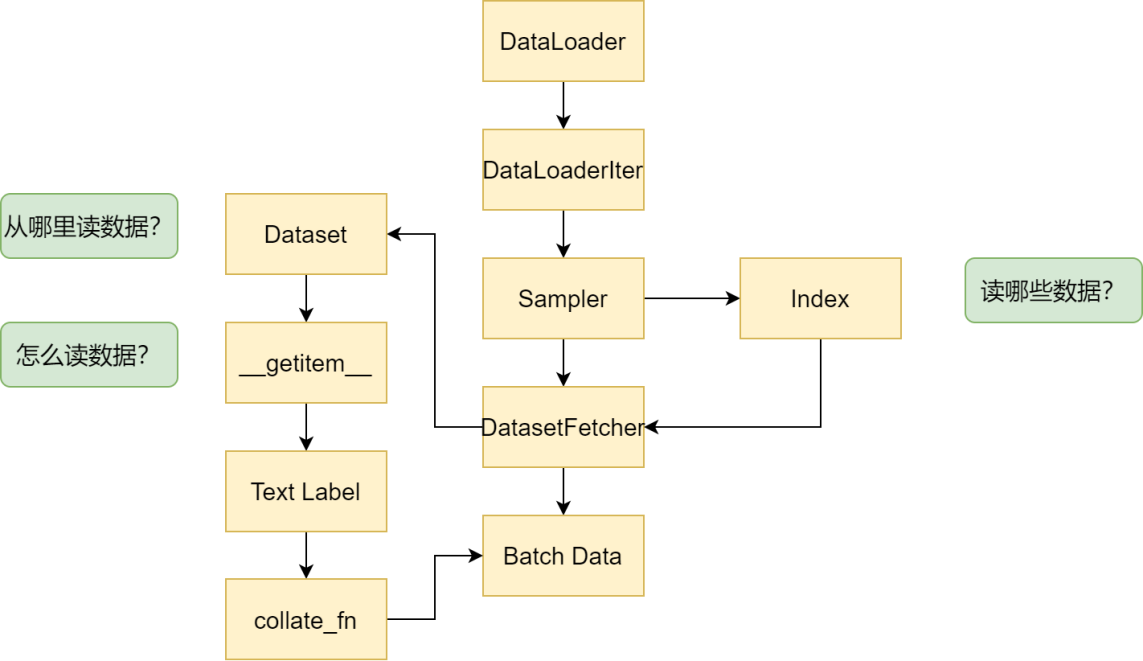
在实践中，主要修改的参数以下标为橙色



顺带介绍一下Epoch、Iteration、Batchsize之间的关系



先看数据读取的主要流程



1. 从DataLoader开始

2. 进入DataLoaderIter，判断单线程还是多线程

3. 进入Sampler进行采样，获得一批一批的索引，这些索引告诉我们需要读取哪些数据、

4. 进入DatasetFetcher，依据索引读取数据

5. Dataset告诉我们数据的地址

6. 自定义的Dataset中会重写\_\_getietm\_\_方法，针对不同的数据来进行定制化的数据读取

7. 到这里就获取的数据的Text和Label

8. 进入collate\_fn将之前获取的个体数据进行组合成batch

9. 一个一个batch组成Batch Data

再来看一个具体的代码

from torch.utils.data import DataLoader

from torch.utils.data.dataset import TensorDataset

# 自构建数据集

dataset = TensorDataset(torch.arange(1, 40))

dl = DataLoader(dataset,

                batch\_size=10,

                shuffle=True,

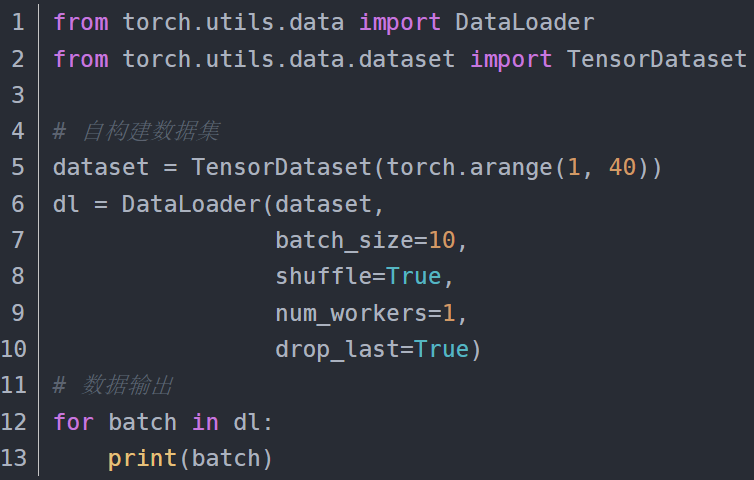
                num\_workers=1,

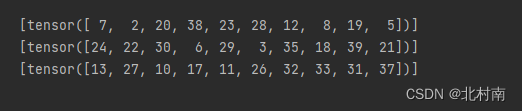
                drop\_last=True)

# 数据输出

for batch in dl:

    print(batch)





因为自定义的数据集只有39条，最后一个batch的数据量小于10，被舍弃掉了

而数据预处理主要是重写Dataset和DataLoader中的方法，因此总体代码如下所示

### **5.5 Pytorch工具**

基于Pytorch已经产生了一些封装完备的工具，而缺点也很明显，数据处理不是很灵活，对于初学者来说，多写代码比较踏实，因此作者不太推荐使用这些方法



## **六、torch.nn**

torch.nn是神经网路工具箱，该工具箱建立于Autograd(主要有自动求导和梯度反向传播功能)，提供了网络搭建的模组，优化器等一系列功能。

搭建一个神经网络模型整个流程是怎么样的呢？

搭建网络流程

1 数据读取

2 定义模型

3 定义损失函数和优化器

4 模型训练

5 获取训练结果

我们拿一个最简单的FNN网络来对经典数据集diabetes糖尿病数据集来进行分类预测，模型性能指标直接采用Loss。

数据集：见博客顶端

​

import numpy as np

import torch

import matplotlib.pyplot as plt

from torch.utils.data import Dataset, DataLoader

# Prepare the dataset

class DiabetesDateset(Dataset):

    # 加载数据集

    def \_\_init\_\_(self, filepath):

        xy = np.loadtxt(filepath, delimiter=',', dtype=np.float32, encoding='utf-8')

        self.len = xy.shape[0]  # shape[0]是矩阵的行数,shape[1]是矩阵的列数

        self.x\_data = torch.from\_numpy(xy[:, :-1])

        self.y\_data = torch.from\_numpy(xy[:, [-1]])

    # 获取数据索引

    def \_\_getitem\_\_(self, index):

        return self.x\_data[index], self.y\_data[index]

    # 获得数据总量

    def \_\_len\_\_(self):

        return self.len

dataset = DiabetesDateset('diabetes.csv')

train\_loader = DataLoader(dataset=dataset, batch\_size=32, shuffle=True, num\_workers=2, drop\_last=True)  # num\_workers为多线程

# Define the model

class FNNModel(torch.nn.Module):

    def \_\_init\_\_(self):

        super(FNNModel, self).\_\_init\_\_()

        self.linear1 = torch.nn.Linear(8, 6)  # 输入数据的特征有8个,也就是有8个维度,随后将其降维到6维

        self.linear2 = torch.nn.Linear(6, 4)  # 6维降到4维

        self.linear3 = torch.nn.Linear(4, 2)  # 4维降到2维

        self.linear4 = torch.nn.Linear(2, 1)  # 2w维降到1维

        self.sigmoid = torch.nn.Sigmoid()  # 可以视其为网络的一层,而不是简单的函数使用

    def forward(self, x):

        x = self.sigmoid(self.linear1(x))

        x = self.sigmoid(self.linear2(x))

        x = self.sigmoid(self.linear3(x))

        x = self.sigmoid(self.linear4(x))

        return x

model = FNNModel()

# Define the criterion and optimizer

criterion = torch.nn.BCELoss(reduction='mean')  # 返回损失的平均值

optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

epoch\_list = []

loss\_list = []

# Training

if \_\_name\_\_ == '\_\_main\_\_':

    for epoch in range(100):

        # i是一个epoch中第几次迭代,一共756条数据,每个mini\_batch为32,所以一个epoch需要迭代23次

        # data获取的数据为(x,y)

        loss\_one\_epoch = 0

        for i, data in enumerate(train\_loader, 0):

            inputs, labels = data

            y\_pred = model(inputs)

            loss = criterion(y\_pred, labels)

            loss\_one\_epoch += loss.item()

            optimizer.zero\_grad()

            loss.backward()

            optimizer.step()

        loss\_list.append(loss\_one\_epoch / 23)

        epoch\_list.append(epoch)

        print('Epoch[{}/{}],loss:{:.6f}'.format(epoch + 1, 100, loss\_one\_epoch / 23))

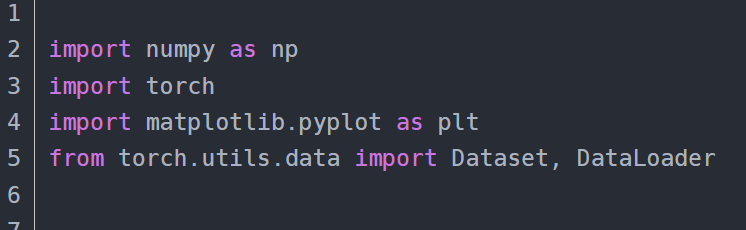
    # Drawing

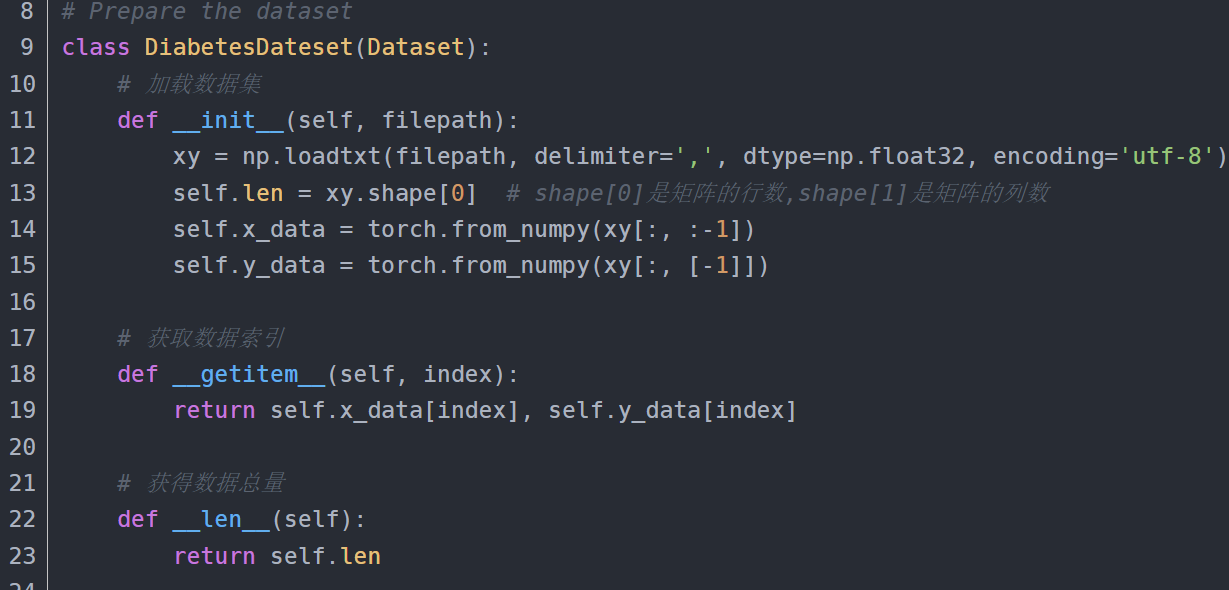
    plt.plot(epoch\_list, loss\_list)

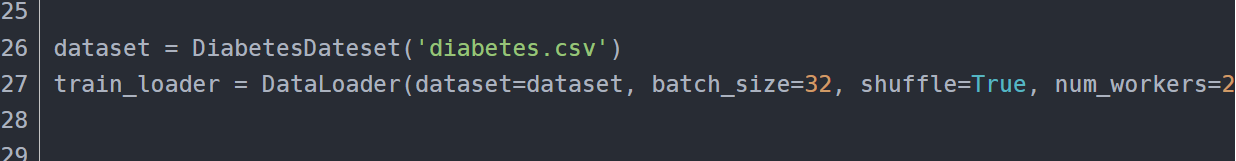
    plt.xlabel('epoch')

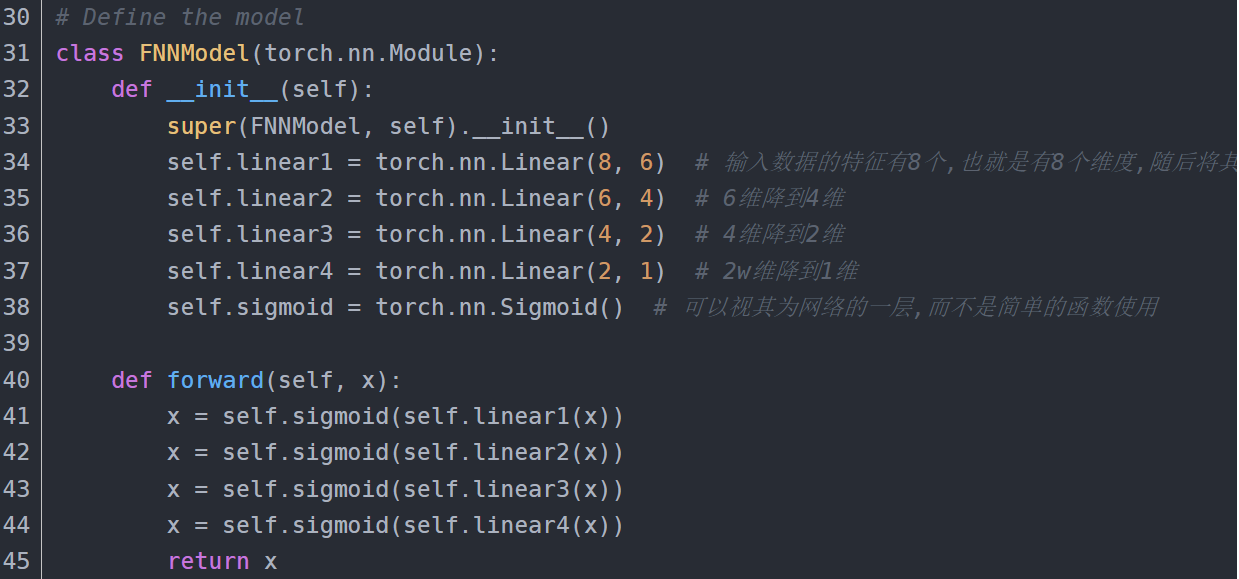
    plt.ylabel('loss')

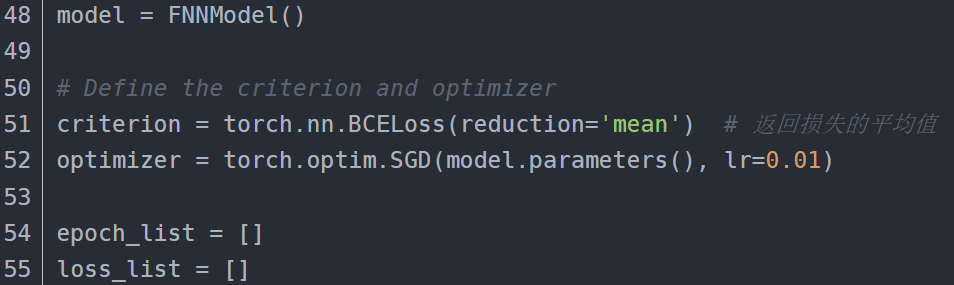
    plt.show()

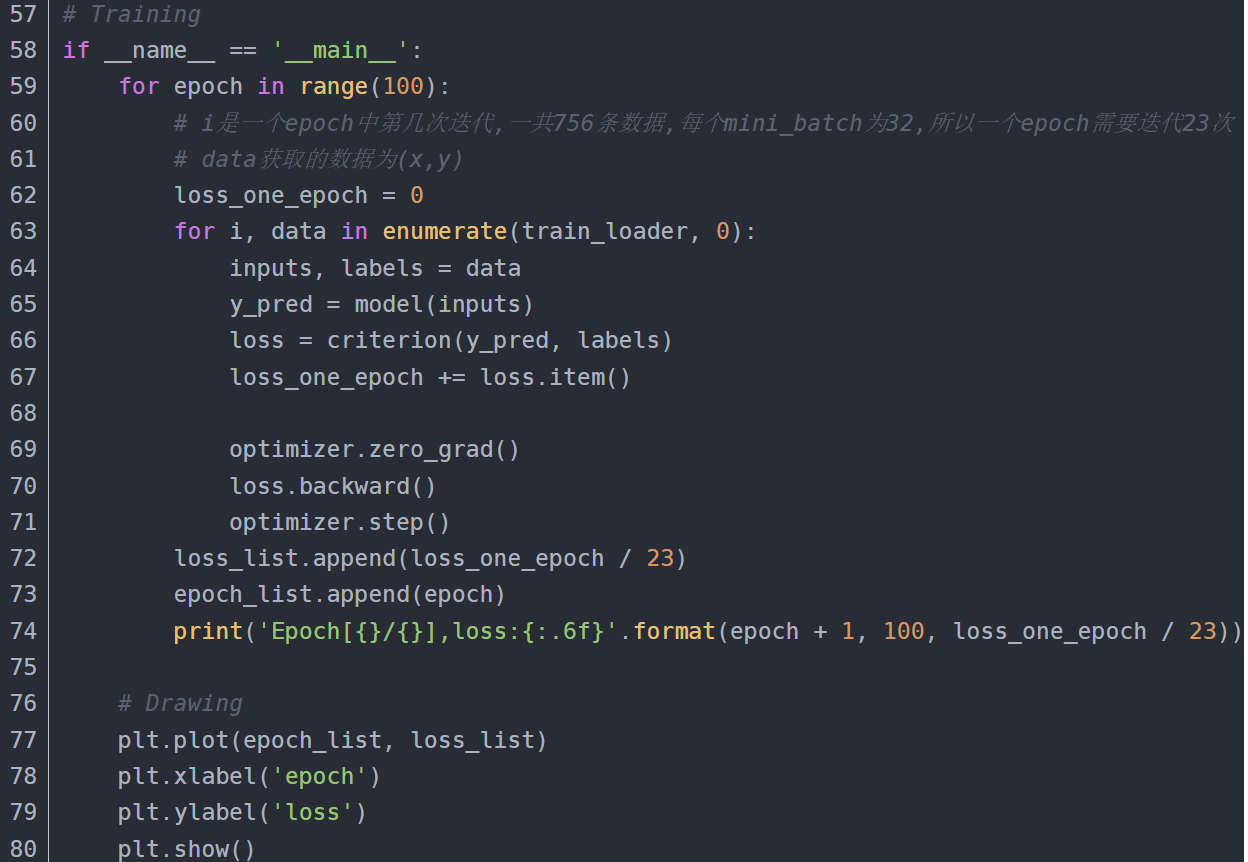
​



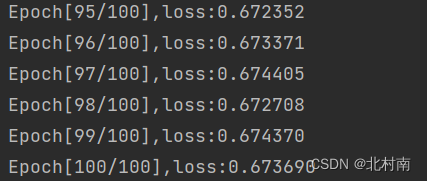


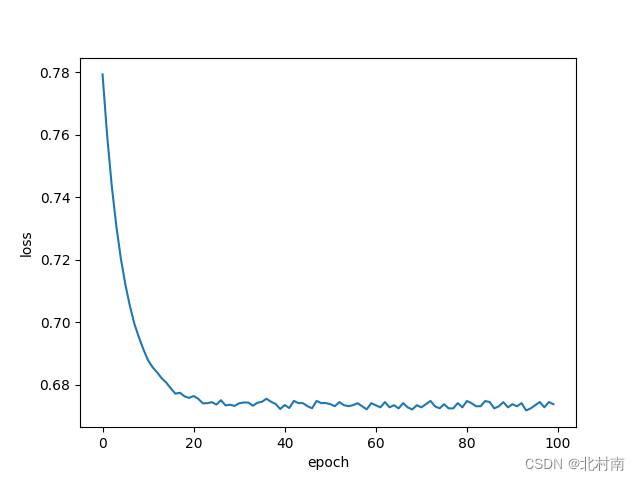






****Result****





注意点

1 创建模型的参数与\_\_init\_\_中的参数一致

2 训练模型的参数与forward中的参数一致

## 参考资料

本文章主要参考Github的各个大牛团队以及Pytorch的官方文档，感谢开源，开源万岁！

1 Chinese-Text-Classfication-Pytorch

2 Pytorch official Chinese documents

3 Pytorch official English doccuments

4 B站 PyTorch深度学习快速入门教程（小土堆）

5 pytorch-tutorial

6 Youtube--Pytorch

7 Pytorch--handbook

8 Deep Learning with PyTorch: A 60 Minute Blitz

以下是作者两个较有诚意的工程，已开源。其中的代码框架结构是我看了较多论文源码后总结出来的一套较为清晰、规范的模板，且我的学术论文都是在此基础上修改的。工程的内容也是热门的CV和NLP领域两个基础性的任务，适合学完Pytorch后的工程实践

Bert/Roberta+Transformer\_BILSTM\_TextCNN实现IMDB文本分类

LeNet、AlexNet、GoogleLeNet、VGG16、ResNet实现COIL20图像分类

————————————————

原文链接：<https://blog.csdn.net/ccaoshangfei/article/details/126074300>

# Ch2.【Pytorch】利用PyTorch实现图像识别

【Pytorch】利用PyTorch实现图像识别

**[紫璇冥](https://www.zhihu.com/people/ren-gong-zhi-neng-fen-xiang" \t "https://zhuanlan.zhihu.com/p/_blank)** 发布于 2023-07-10 14:17・IP 属地江苏

https://zhuanlan.zhihu.com/p/642603503

使用torchvision库的datasets类加载常用的数据集或自定义数据集

图像识别是计算机视觉中的一个基础任务，它的目标是让计算机能够识别图像中的物体、场景或者概念，并将它们分配到预定义的类别中。例如，给定一张猫的图片，图像识别系统应该能够输出“猫”这个类别。

为了训练和评估图像识别系统，我们需要有大量的带有标注的图像数据集。常用的图像分类数据集有：

* ImageNet：一个包含超过1400万张图片和2万多个类别的大型数据库，是目前最流行和最具挑战性的图像分类基准之一。
* CIFAR-10/CIFAR-100：一个包含6万张32×32大小的彩色图片和10或100个类别的小型数据库，适合入门级和快速实验。
* MNIST：一个包含7万张28×28大小的灰度手写数字图片和10个类别的经典数据库，是深度学习中最常用的测试集之一。
* Fashion-MNIST：一个包含7万张28×28大小的灰度服装图片和10个类别的数据库，是MNIST数据库在时尚领域上更加复杂和现代化版本。

使用torchvision库可以方便地加载这些常用数据集或者自定义数据集。torchvision.datasets提供了一些加载数据集或者下载数据集到本地缓存文件夹（默认为./data）并返回Dataset对象（torch.utils.data.Dataset） 的函数。Dataset对象可以存储样本及其对应标签，并提供索引方式（dataset[i]）来获取第i个样本。例如，要加载CIFAR-10训练集并进行随机打乱，可以使用以下代码：

import torchvision

import torchvision.transforms as transforms

transform = transforms.Compose([transforms.ToTensor()]) # 定义转换函数，将PIL.Image转换为torch.Tensor

trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform) # 加载CIFAR-10训练集

trainloader = torch.utils.data.DataLoader(trainset, batch\_size=4, shuffle=True) # 定义DataLoader对象，用于批量加载数据

## **使用torchvision库进行数据增强和变换，自定义自己的图像分类数据集并使用torchvision库加载它们**

数据增强和变换：为了提高模型的[泛化能力](https://zhida.zhihu.com/search?q=%E6%B3%9B%E5%8C%96%E8%83%BD%E5%8A%9B&zhida_source=entity&is_preview=1" \t "https://zhuanlan.zhihu.com/p/_blank)和数据利用率，我们通常会对图像数据进行一些随机的变换，例如裁剪、旋转、翻转、缩放、亮度调整等。这些变换可以在一定程度上模拟真实场景中的图像变化，增加模型对不同视角和光照条件下的物体识别能力。torchvision.transforms提供了一些常用的图像变换函数，可以组合成一个transform对象，并传入datasets类中作为参数。例如，要对CIFAR-10训练集进行随机水平翻转和随机裁剪，并将[图像归一化](https://zhida.zhihu.com/search?q=%E5%9B%BE%E5%83%8F%E5%BD%92%E4%B8%80%E5%8C%96&zhida_source=entity&is_preview=1" \t "https://zhuanlan.zhihu.com/p/_blank)到[-1, 1]范围内，可以使用以下代码：

import torchvision

import torchvision.transforms as transforms

transform = transforms.Compose([

transforms.RandomHorizontalFlip(), # 随机水平翻转

transforms.RandomCrop(32, padding=4), # 随机裁剪到32×32大小，并在边缘填充4个像素

transforms.ToTensor(), # 将PIL.Image转换为torch.Tensor

transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)) # 将RGB三个通道的值归一化到[-1, 1]范围内

])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform) # 加载CIFAR-10训练集，并应用上述变换

trainloader = torch.utils.data.DataLoader(trainset, batch\_size=4, shuffle=True) # 定义DataLoader对象，用于批量加载数据

自定义图像分类数据集：如果我们有自己的图像分类数据集，我们可以通过继承torch.utils.data.Dataset类来自定义一个Dataset对象，并实现\_\_len\_\_和\_\_getitem\_\_两个方法。\_\_len\_\_方法返回数据集中样本的数量，\_\_getitem\_\_方法根据给定的索引返回一个样本及其标签。例如，假设我们有一个文件夹结构如下：

my\_dataset/

├── class\_0/

│ ├── image\_000.jpg

│ ├── image\_001.jpg

│ └── ...

├── class\_1/

│ ├── image\_000.jpg

│ ├── image\_001.jpg

│ └── ...

└── ...

其中每个子文件夹代表一个类别，每个子文件夹中包含该类别对应的图像文件。我们可以使用以下代码来自定义一个Dataset对象，并加载这个数据集：

import torch.utils.data as data

from PIL import Image

import os

class MyDataset(data.Dataset):

def \_\_init\_\_(self, root\_dir, transform=None):

self.root\_dir = root\_dir # 根目录路径

self.transform = transform # 变换函数

self.classes = sorted(os.listdir(root\_dir)) # 类别列表（按字母顺序排序）

self.class\_to\_idx = {c: i for i,c in enumerate(self.classes)} # 类别名到索引的映射

self.images = [] # 图片路径列表（相对于根目录）

self.labels = [] # 标签列表（整数）

for c in self.classes:

c\_dir = os.path.join(root\_dir, c) # 类别子目录路径

for img\_name in sorted(os.listdir(c\_dir)): # 遍历每个图片文件名（按字母顺序排序）

img\_path = os.path.join(c,img\_name) # 图片相对路径（相对于根目录）

label = self.class\_to\_idx[c] # 图

## **使用torchvision库的models类加载预训练模型或自定义模型**

加载预训练模型或自定义模型：torchvision.models提供了一些常用的图像分类模型，例如AlexNet、VGG、ResNet等，并且可以选择是否加载在ImageNet数据集上预训练好的权重。这些模型可以直接用于图像分类任务，也可以作为特征提取器或者微调（fine-tune）的基础。例如，要加载一个预训练好的ResNet-18模型，并冻结除最后一层外的所有参数，可以使用以下代码：

import torchvision.models as models

model = models.resnet18(pretrained=True) # 加载预训练好的ResNet-18模型

for param in model.parameters(): # 遍历所有参数

param.requires\_grad = False # 将参数的梯度设置为False，表示不需要更新

num\_features = model.fc.in\_features # 获取[全连接层](https://zhida.zhihu.com/search?q=%E5%85%A8%E8%BF%9E%E6%8E%A5%E5%B1%82&zhida_source=entity&is_preview=1" \t "https://zhuanlan.zhihu.com/p/_blank)（fc）的输入特征数

model.fc = torch.nn.Linear(num\_features, 10) # 替换全连接层为一个新的线性层，输出特征数为10（假设有10个类别）

如果我们想要自定义自己的图像分类模型，我们可以通过继承torch.nn.Module类来实现一个Module对象，并实现\_\_init\_\_和forward两个方法。\_\_init\_\_方法用于定义模型中需要的各种层和参数，forward方法用于定义前向传播过程。例如，要自定义一个简单的[卷积神经网络](https://zhida.zhihu.com/search?q=%E5%8D%B7%E7%A7%AF%E7%A5%9E%E7%BB%8F%E7%BD%91%E7%BB%9C&zhida_source=entity&is_preview=1" \t "https://zhuanlan.zhihu.com/p/_blank)（CNN）模型，可以使用以下代码：

import torch.nn as nn

class MyCNN(nn.Module):

def \_\_init\_\_(self):

super(MyCNN, self).\_\_init\_\_() # 调用父类构造函数

self.conv1 = nn.Conv2d(3, 6, 5) # 定义第一个卷积层，输入通道数为3（RGB），输出通道数为6，卷积核大小为5×5

self.pool = nn.MaxPool2d(2, 2) # 定义最大池化层，池化核大小为2×2，步长为2

self.conv2 = nn.Conv2d(6, 16, 5) # 定义第二个卷积层，输入通道数为6，输出通道数为16，卷积核大小为5×5

self.fc1 = nn.Linear(16 \* 5 \* 5, 120) # 定义第一个全连接层，输入特征数为16×5×5（根据卷积和池化后的图像大小计算得到），输出特征数为120

self.fc2 = nn.Linear(120, 84) # 定义第二个全连接层，输入特征数为120，输出特征数为84

self.fc3 = nn.Linear(84, 10) # 定义第三个全连接层，输入特征数为84，

## **forward方法**

forward方法用于定义前向传播过程，即如何根据输入的图像张量（Tensor）计算出输出的类别概率分布。我们可以使用定义好的各种层和参数，并结合一些[激活函数](https://zhida.zhihu.com/search?q=%E6%BF%80%E6%B4%BB%E5%87%BD%E6%95%B0&zhida_source=entity&is_preview=1" \t "https://zhuanlan.zhihu.com/p/_blank)（如ReLU）和归一化函数（如softmax）来实现forward方法。例如，要实现上面自定义的CNN模型的forward方法，可以使用以下代码：

import torch.nn.functional as F

class MyCNN(nn.Module):

def \_\_init\_\_(self):

# 省略\_\_init\_\_方法的内容

...

def forward(self, x): # 定义前向传播过程，x是输入的图像张量

x = self.pool(F.relu(self.conv1(x))) # 将x通过第一个卷积层和ReLU激活函数，然后通过最大池化层

x = self.pool(F.relu(self.conv2(x))) # 将x通过第二个卷积层和ReLU激活函数，然后通过最大池化层

x = x.view(-1, 16 \* 5 \* 5) # 将x展平为一维向量，-1表示自动推断批量大小

x = F.relu(self.fc1(x)) # 将x通过第一个全连接层和ReLU激活函数

x = F.relu(self.fc2(x)) # 将x通过第二个全连接层和ReLU激活函数

x = self.fc3(x) # 将x通过第三个全连接层

x = F.softmax(x, dim=1) # 将x通过softmax函数，沿着第一个维度（类别维度）进行归一化，得到类别概率分布

return x # 返回输出的类别概率分布

## **进行模型训练和测试，使用matplotlib.pyplot库可视化结果**

模型训练和测试是机器学习中的重要步骤，它们可以帮助我们评估模型的性能和泛化能力。matplotlib.pyplot是一个Python库，它可以用来绘制各种类型的图形，包括曲线图、散点图、直方图等。使用matplotlib.pyplot库可视化结果的一般步骤如下：

* 导入matplotlib.pyplot模块，并设置一些参数，如字体、分辨率等。
* 创建一个或多个图形对象（figure），并指定大小、标题等属性。
* 在每个图形对象中创建一个或多个子图（subplot），并指定位置、坐标轴等属性。
* 在每个子图中绘制数据，使用不同的函数和参数，如plot、scatter、bar等。
* 添加一些修饰元素，如图例（legend）、标签（label）、标题（title）等。
* 保存或显示图形。

例如：使用matplotlib.pyplot库绘制了一个[线性回归模型](https://zhida.zhihu.com/search?q=%E7%BA%BF%E6%80%A7%E5%9B%9E%E5%BD%92%E6%A8%A1%E5%9E%8B&zhida_source=entity&is_preview=1" \t "https://zhuanlan.zhihu.com/p/_blank)的训练误差和测试误差曲线

# 导入模块

import matplotlib.pyplot as plt

import numpy as np

# 设置字体和分辨率

plt.rcParams["font.sans-serif"] = ["SimHei"]

plt.rcParams["axes.unicode\_minus"] = False

%config InlineBackend.figure\_format = "retina"

# 生成数据

x = np.linspace(0, 10, 100)

y = 3 \* x + 5 + np.random.randn(100) \* 2 # 真实值

w = np.random.randn() # 随机初始化权重

b = np.random.randn() # 随机初始化偏置

# 定义[损失函数](https://zhida.zhihu.com/search?q=%E6%8D%9F%E5%A4%B1%E5%87%BD%E6%95%B0&zhida_source=entity&is_preview=1" \t "https://zhuanlan.zhihu.com/p/_blank)

def loss(y\_true, y\_pred):

return ((y\_true - y\_pred) \*\* 2).mean()

# 定义梯度下降函数

def gradient\_descent(x, y\_true, w, b, lr):

y\_pred = w \* x + b # 预测值

dw = -2 \* (x \* (y\_true - y\_pred)).mean() # 权重梯度

db = -2 \* (y\_true - y\_pred).mean() # 偏置梯度

w = w - lr \* dw # 更新权重

b = b - lr \* db # 更新偏置

return w, b

# 训练模型，并记录每轮的训练误差和测试误差

epochs = 20 # 训练轮数

lr = 0.01 # 学习率

train\_loss\_list = [] # 训练误差列表

test\_loss\_list = [] # 测试误差列表

for epoch in range(epochs):

# 划分训练集和测试集（8:2）

train\_index = np.random.choice(100, size=80, replace=False)

test\_index = np.setdiff1d(np.arange(100), train\_index)

x\_train, y\_train = x[train\_index], y[train\_index]

x\_test, y\_test = x[test\_index], y[test\_index]

# 梯度下降更新参数，并计算训练误差和测试误差

w, b = gradient\_descent(x\_train, y\_train, w, b, lr)

train\_loss = loss(y\_train, w \* x\_train + b)

test\_loss = loss(y\_test, w \* x\_test + b)

# 打印结果，并将误差添加到列表中

print(f"Epoch {epoch+1}, Train Loss: {train\_loss:.4f}, Test Loss: {test\_loss:.4f}")

train\_loss\_list.append(train\_loss)

test\_loss\_list.append(test\_loss)

# 创建一个图形对象，并设置大小为8\*6英寸

plt.figure(figsize=(8,6))

# 在图形对象中创建一个子图，并设置位置为1行1列的第1个

plt.subplot(1, 1, 1)

# 在子图中绘制训练误差和测试误差曲线，使用不同的颜色和标签

plt.plot(np.arange(epochs), train\_loss\_list, "r", label="Train Loss")

plt.plot(np.arange(epochs), test\_loss\_list, "b", label="Test Loss")

# 添加图例、坐标轴标签和标题

plt.legend()

plt.xlabel("Epoch")

plt.ylabel("Loss")

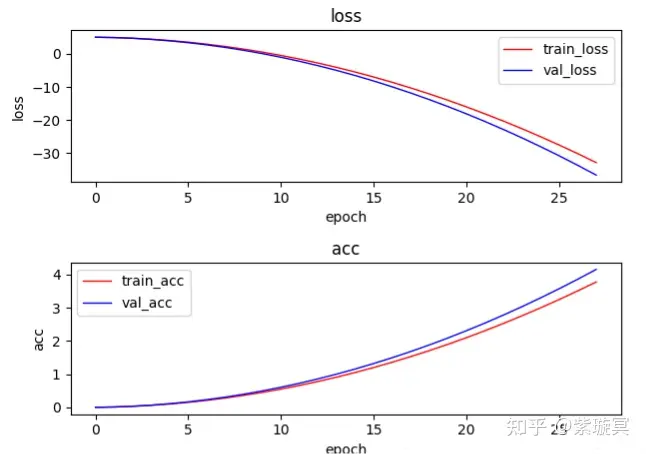
plt.title("Linear Regression Loss Curve")

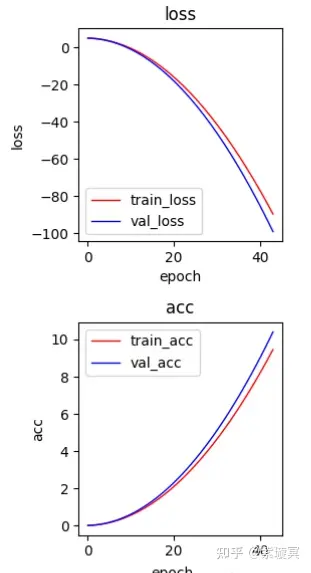
# 保存或显示图形

#plt.savefig("loss\_curve.png")

plt.show()

运行后，可以看到如下的图形：





参考： PyTorch官方网站

# Ch3 《动手学深度学习》第二版

<https://zh-v2.d2l.ai/index.html>

强烈推荐！！李沐老师《动手学深度学习》最新Pytorch版！

目录

前言

安装

符号

1.引言

1.1.日常生活中的机器学习

1.2.机器学习中的关键组件

1.3.各种机器学习问题

1.4.起源

1.5.深度学习的发展

1.6.深度学习的成功案例

1.7.特点

1.8.小结

1.9.练习

2.预备知识

2.1.数据操作

2.2.数据预处理

2.3.线性代数

2.4.微积分

2.5.自动微分

2.6.概率

3.线性神经网络

3.1.线性回归

3.2.线性回归的从零开始实现

3.3.线性回归的简洁实现

3.4. softmax回归

3.5.图像分类数据集

3.6.softmax回归的从零开始实现

3.7.softmax回归的简洁实现

4.多层感知机

4.1.多层感知机

4.2.多层感知机的从零开始实现

4.3.多层感知机的简洁实现

4.4.模型选择、欠拟合和过拟合

4.5.权重衰减

4.6.暂退法(Dropout)

4.7.前向传播、反向传播和计算图

4.8.数值稳定性和模型初始化

4.9.环境和分布偏移

4.10.实战Kaggle比赛:预测房价

5.深度学习计算

5.1.层和块

5.2.参数管理

5.3.延后初始化

5.4.自定义层

6.卷积神经网络

6.1.从全连接层到卷积

6.2.图像卷积

6.3.填充和步幅

6.4.多输入多输出通道

6.5. 汇聚层

6.6.卷积神经网络(LeNet)

7.现代卷积神经网络

7.1.深度卷积神经网络(AlexNet)

7.2.使用块的网络(VGG)

7.3.网络中的网络

(NiN)

7.4.含并行连结的网络(GoogLeNet)

7.5.批量规范化

7.6.残差网络(ResNet)

7.7.稠密连接网络(DenseNet)

8.循环神经网络

8.1.序列模型

8.2.文本预处理

8.3.语言模型和数据集

8.4.循环神经网络

8.5.循环神经网络的从零开始实现

8.6.循环神经网络的简洁实现

9.现代循环神经网络

9.1.门控循环单元(GRU)

9.2.长短期记忆网络(LSTM)

9.3.深度循环神经网络

9.4.双向循环神经网络

9.5.机器翻译与数据集

9.6.编码器-解码器架构

9.7.序列到序列学习(seq2seq)

9.8.束搜索

10.注意力机制

10.1.注意力提示

10.2.注意力汇聚:Nadaraya-Watson 核回归

10.3.注意力评分函数

10.4.Bahdanau 注意力

10.5.多头注意力

10.6.自注意力和位置编码

10.7.Transformer

11.优化算法

11.1.优化和深度学习

11.2.凸性

11.3.梯度下降

11.4.随机梯度下降

12.计算性能

12.1.编译器和解释器

12.2.异步计算

12.3.自动并行

12.4.硬件

12.5.多GPU训练

12.6.多GPU的简洁实现

12.7.参数服务器

13.计算机视觉

13.1.图像增广

13.2.微调

13.3.目标检测和边界框

13.4.锚框

13.5.多尺度目标检测

13.6.目标检测数据集

13.7.单发多框检测(SSD)

13.8.区域卷积神经网络(R-CNN)系列

13.9.语义分割和数据集

13.10.转置卷积

13.11.全卷积网络

13.12.风格迁移

13.13.实战 Kaggle 比赛:图像分类(CIFAR-10)

# Ch4.用golang部署pytorch模型

百度AI智能回答：

在Go中部署PyTorch模型通常涉及到调用PyTorch提供的预测接口。这可以通过几种方法实现，例如使用CGo直接调用Python代码，或者使用标准的进程间通信（IPC）方法，例如gRPC或REST API。

以下是一个简化的例子，使用CGo和Python API在Go中部署PyTorch模型：

首先，你需要确保你的Python环境中有你的PyTorch模型和相应的接口代码。例如，你可能有一个model.py文件，它定义了一个可以加载模型和进行预测的函数。

# Ch5.手机可部署的 ai模型格式

手机可部署的AI模型格式通常取决于手机的操作系统、硬件性能以及所选择的AI框架。以下是一些常见的手机可部署的AI模型格式：

### **一、开源模型格式**

1.ONNX（Open Neural Network Exchange）：

1.1ONNX是一个开放格式，用于表示深度学习模型。它使得模型可以在不同的框架、工具、运行时之间移植。

1.2许多流行的深度学习框架都支持导出为ONNX格式，包括PyTorch、TensorFlow等。

1.3手机上的一些AI推理引擎（如ONNX Runtime）可以加载并运行ONNX模型。

2.Core ML（Apple Core Machine Learning）：

2.1 Core ML是Apple推出的一种用于在iOS设备上运行机器学习模型的格式。

2.2它支持将训练好的模型从各种框架（如TensorFlow、PyTorch等）转换为Core ML格式，并在iOS应用中直接运行。

2.3 Core ML模型可以利用iOS设备的硬件加速功能（如GPU和Neural Engine）来提高推理速度。

3.TFLite（TensorFlow Lite）：

3.1 TFLite是TensorFlow的轻量级解决方案，专为移动设备和嵌入式设备设计。

3.2 它可以将TensorFlow模型转换为高效的TFLite格式，以便在这些设备上运行。

3.3 TFLite模型支持硬件加速，并提供了优化的内存管理和性能。

****4.PyTorch Mobile****：

4.1 PyTorch Mobile是PyTorch的移动端版本，它允许开发者将PyTorch模型直接部署到移动设备上。

4.2 PyTorch Mobile支持动态计算和自动微分，使得模型可以在移动设备上实时更新和训练（尽管这通常不是移动部署的常见用例）。

4.3 它还提供了与PyTorch生态系统中的其他工具的兼容性，如TorchScript。

### **二、其他模型格式**

除了上述开源模型格式外，还有一些特定的模型格式是为特定平台或框架设计的。例如：

* Caffe2：由Facebook开发，后来与PyTorch集成，但Caffe2模型格式仍然可以在一些移动设备上运行。
* TVM：一个开源的深度学习编译栈，它可以将模型编译为在各种硬件上高效运行的格式，包括移动设备。
* 其他专有格式：一些公司或组织可能开发了自己的专有模型格式，用于在特定设备上运行。

### **三、部署注意事项**

* 硬件兼容性：不同的手机型号和硬件配置可能支持不同的模型格式和推理引擎。在选择模型格式时，请确保它与目标设备的硬件兼容。
* 性能优化：为了在手机设备上获得最佳的推理性能，请考虑使用硬件加速功能（如GPU、NPU等）并优化模型结构。
* 模型大小：手机设备的存储空间有限，因此请确保所选的模型格式能够生成较小的模型文件，以便节省存储空间。
* 隐私和安全：在部署AI模型时，请确保遵守相关的隐私和安全法规，并采取措施保护用户数据的安全性和隐私性。

综上所述，手机可部署的AI模型格式有多种选择，具体取决于目标设备的硬件性能、操作系统以及所选的AI框架。在选择模型格式时，请务必考虑上述因素以确保最佳的部署效果。

# Ch6.ncnn

ncnn 是一个为手机端优化的高性能神经网络前向计算框架。它支持跨平台，已经在 Android、iOS、Linux（x86/arm）、Windows（x86/x64） 上进行了测试，并且具有良好的稳定性和速度表现。ncnn 从设计之初就深刻考虑手机端的部署和使用，因此它无需依赖任何第三方库，所有代码均为 C++ 编写，不依赖于任何平台的 API，能够轻松集成到各种项目中。

ncnn 的特点包括：

* 高性能：通过针对手机端的优化，ncnn 能够实现高效的神经网络前向计算。
* 跨平台：支持多种操作系统和硬件平台，无需针对每个平台进行额外的开发。
* 易于集成：由于不依赖任何第三方库，ncnn 可以轻松地集成到各种项目中，无论是原生应用还是跨平台应用。
* 丰富的模型支持：ncnn 支持从多种深度学习框架（如 TensorFlow、PyTorch、Caffe 等）导出的模型，并且提供了转换工具来将这些模型转换为 ncnn 可以识别的格式。
* 易用性：ncnn 提供了简洁的 API，使得开发者可以轻松地加载模型、进行推理以及获取结果。
* 定制化：ncnn 允许开发者根据自己的需求进行定制化的优化，如调整参数、修改网络结构等。

要使用 ncnn 在手机端部署 AI 模型，通常需要经过以下步骤：

* 模型转换：首先，你需要将训练好的深度学习模型从原始框架（如 TensorFlow、PyTorch 等）转换为 ncnn 可以识别的格式。ncnn 提供了转换工具，可以自动完成这一步。
* 集成 ncnn：将 ncnn 框架集成到你的手机项目中。这通常包括将 ncnn 的源代码或预编译库添加到你的项目中，并配置相关的编译选项。
* 加载模型：在你的代码中，使用 ncnn 提供的 API 加载转换后的模型文件。
* 进行推理：将输入数据传递给模型，进行前向计算，并获取输出结果。
* 处理结果：根据输出结果，执行相应的操作，如显示结果、保存结果等。

总之，ncnn 是一个专为手机端设计的高性能神经网络前向计算框架，它支持跨平台、易于集成、性能优越，并且提供了丰富的模型支持和定制化的优化选项。通过使用 ncnn，你可以轻松地在手机端部署 AI 模型，并实现高效的推理计算。

# Ch7.探索ONNX-Go 将深度学习模型部署到Go应用程序中

【山东大学软件学院 21 级项目实训】探索ONNX-Go：将深度学习模型部署到Go应用程序中

篠原春奈OvO 于 2024-06-16 23:28:10 发布

原文链接：<https://blog.csdn.net/u012185664/article/details/139728594>

随着深度学习技术的快速发展，许多开发人员希望能够将训练好的模型集成到他们的应用程序中。ONNX（Open Neural Network Exchange）作为一种开放的深度学习模型表示格式，为不同深度学习框架之间的模型转换和交换提供了标准化的解决方案。在本文中，我们将重点介绍如何使用 ONNX-Go 这一 Go 语言库，将训练好的深度学习模型轻松地部署到 Go 应用程序中。

什么是ONNX-Go？

ONNX-Go 是一个基于 Go 语言的开源库，旨在帮助开发人员在 Go 应用程序中加载、运行和推理 ONNX 格式的深度学习模型。该库提供了一组功能强大的工具，使开发人员能够轻松地将训练好的深度学习模型集成到他们的 Go 项目中，从而实现各种机器学习任务。

如何使用ONNX-Go？

步骤一：安装ONNX-Go

首先安装 ONNX-Go 库。可以通过 Go 模块管理工具来安装 ONNX-Go，具体步骤如下：

go get -u github.com/owulveryck/onnx-go

步骤二：加载模型

接下来，加载训练好的 ONNX 格式的深度学习模型。ONNX-Go 提供了简单易用的 API 来加载模型，示例代码如下：

package main

import (

"github.com/owulveryck/onnx-go"

)

func main() {

modelPath := "path/to/your/model.onnx"

model := onnx.NewModel(modelPath)

if err := model.Load(); err != nil {

panic(err)

}

}

步骤三：进行推理

一旦模型加载成功，就可以使用 ONNX-Go 进行推理。下面是一个简单的示例代码，演示如何使用 ONNX-Go 进行推理：

package main

import (

"github.com/owulveryck/onnx-go"

)

func main() {

// 加载模型

modelPath := "path/to/your/model.onnx"

model := onnx.NewModel(modelPath)

if err := model.Load(); err != nil {

panic(err)

}

// 准备输入数据

input := []float32{1.0, 2.0, 3.0}

// 进行推理

output, err := model.Predict(input)

if err != nil {

panic(err)

}

// 处理输出

// TODO: 处理输出数据

}