一、Pytorch的建模流程

使用Pytorch实现神经网络模型的一般流程包括：

1，准备数据

2，定义模型

6大步骤

3，训练模型

4，评估模型

5，使用模型

6，保存模型。

二、Pytorch的核心概念

Pytorch是一个基于Python的机器学习库。它广泛应用于计算机视觉，自然语言处理等深度学习领域。是目前和TensorFlow分庭抗礼的深度学习框架，在学术圈颇受欢迎。

它主要提供了以下两种核心功能：

1，支持GPU加速的张量计算。

2，方便优化模型的自动微分机制。

Pytorch的主要优点：

- 简洁易懂：Pytorch的API设计的相当简洁一致。基本上就是tensor, autograd, nn三级封装。学习起来非常容易。有一个这样的段子，说TensorFlow的设计哲学是 Make it complicated, Keras 的设计哲学是 Make it complicated and hide it, 而Pytorch的设计哲学是 Keep it simple and stupid.
- 便于调试：Pytorch采用动态图，可以像普通Python代码一样进行调试。不同于TensorFlow, Pytorch的报错说明通常很容易看懂。有一个这样的段子，说你永远不可能从TensorFlow的报错说明中找到它出错的原因。
- 强大高效：Pytorch提供了非常丰富的模型组件，可以快速实现想法。并且运行速度很快。目前大部分深度学习相关的Paper都是用Pytorch实现的。有些研究人员表示，从使用TensorFlow转换为使用Pytorch之后，他们的睡眠好多了，头发比以前浓密了，皮肤也比以前光滑了。

三、Pytorch的层次结构

本章我们介绍Pytorch中5个不同的层次结构：即硬件层，内核层，低阶API，中阶API，高阶API torchkeras【torchkeras】。并以线性回归和DNN二分类模型为例，直观对比展示在不同层级实现模型的特点。

Pytorch的层次结构从低到高可以分成如下五层。

最底层为硬件层，Pytorch支持CPU、GPU加入计算资源池。

第二层为C++实现的内核。

第三层为Python实现的操作符，提供了封装C++内核的低级API指令，主要包括各种张量操作算子、自动微分、变量管理.
如torch.tensor,torch.cat,torch.autograd.grad,nn.Module.
如果把模型比作一个房子，那么第三层API就是【模型之砖】。

第四层为Python实现的模型组件，对低级API进行了函数封装，主要包括各种模型层，损失函数，优化器，数据管道等等。
如torch.nn.Linear,torch.nn.BCE,torch.optim.Adam,torch.utils.data.DataLoader.
如果把模型比作一个房子，那么第四层API就是【模型之墙】。

第五层为Python实现的模型接口。Pytorch没有官方的高阶API。为了便于训练模型，作者仿照keras中的模型接口，使用了不到300行代码，封装了Pytorch的高阶模型接口torchkeras.Model。如果把模型比作一个房子，那么第五层API就是模型本身，即【模型之屋】。

四、Pytorch的低阶API

Pytorch的低阶API主要包括张量操作，动态计算图和自动微分。

如果把模型比作一个房子，那么低阶API就是【模型之砖】。

在低阶API层次上，可以把Pytorch当做一个增强版的numpy来使用。

Pytorch提供的方法比numpy更全面，运算速度更快，如果需要的话，还可以使用GPU进行加速。

前面几章我们对低阶API已经有了一个整体的认识，本章我们将重点详细介绍张量操作和动态计算图。

张量的操作主要包括张量的结构操作和张量的数学运算。

张量结构操作诸如：张量创建，索引切片，维度变换，合并分割。

张量数学运算主要有：标量运算，向量运算，矩阵运算。另外我们会介绍张量运算的广播机制。

动态计算图我们将主要介绍动态计算图的特性，计算图中的Function，计算图与反向传播。

五、Pytorch的中阶API

我们将主要介绍Pytorch的如下中阶API

- 数据管道
- 模型层
- 损失函数
- TensorBoard可视化

如果把模型比作一个房子，那么中阶API就是【模型之墙】。

六、Pytorch的高阶API

Pytorch没有官方的高阶API。一般通过nn.Module来构建模型并编写自定义训练循环。

为了更加方便地训练模型，作者编写了仿keras的Pytorch模型接口：torchkeras， 作为Pytorch的高阶API。

本章我们主要详细介绍Pytorch的高阶API如下相关的内容。

- 构建模型的3种方法(继承nn.Module基类，使用nn.Sequential，辅助应用模型容器)
- 训练模型的3种方法(脚本风格，函数风格，torchkeras.Model类风格)
- 使用GPU训练模型(单GPU训练，多GPU训练)

1-1,结构化数据建模流程范例

```
import os
import datetime

#打印时间
def printbar():
    nowtime = datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S')
    print("\n"+"=========="*8 + "%s"%nowtime)

#mac系统上pytorch和matplotlib在jupyter中同时跑需要更改环境变量
os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"
```

一，准备数据

titanic数据集的目标是根据乘客信息预测他们在Titanic号撞击冰山沉没后能否生存。

结构化数据一般会使用Pandas中的DataFrame进行预处理。

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import torch
from torch import nn
from torch.utils.data import Dataset,DataLoader,TensorDataset

dftrain_raw = pd.read_csv('./data/titanic/train.csv')
dftest_raw = pd.read_csv('./data/titanic/test.csv')
dftrain_raw.head(10)
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	493	0	1	Molson, Mr. Harry Markland	male	55.0	0	0	113787	30.5000	C30	S
1	53	1	1	Harper, Mrs. Henry Sleeper (Myna Haxtun)	female	49.0	1	0	PC 17572	76.7292	D33	C
2	388	1	2	Buss, Miss. Kate	female	36.0	0	0	27849	13.0000	NaN	S
3	192	0	2	Carbines, Mr. William	male	19.0	0	0	28424	13.0000	NaN	S
4	687	0	3	Panula, Mr. Jaako Arnold	male	14.0	4	1	3101295	39.6875	NaN	S
5	16	1	2	Hewlett, Mrs. (Mary D Kingcome)	female	55.0	0	0	248706	16.0000	NaN	S
6	228	0	3	Lovell, Mr. John Hall ("Henry")	male	20.5	0	0	A/5 21173	7.2500	NaN	S
7	884	0	2	Banfield, Mr. Frederick James	male	28.0	0	0	C.A./SOTON 34068	10.5000	NaN	S
8	168	0	3	Skoog, Mrs. William (Anna Bernhardina Karlsson)	female	45.0	1	4	347088	27.9000	NaN	S
9	752	1	3	Moor, Master. Meier	male	6.0	0	1	392096	12.4750	E121	S

字段说明:

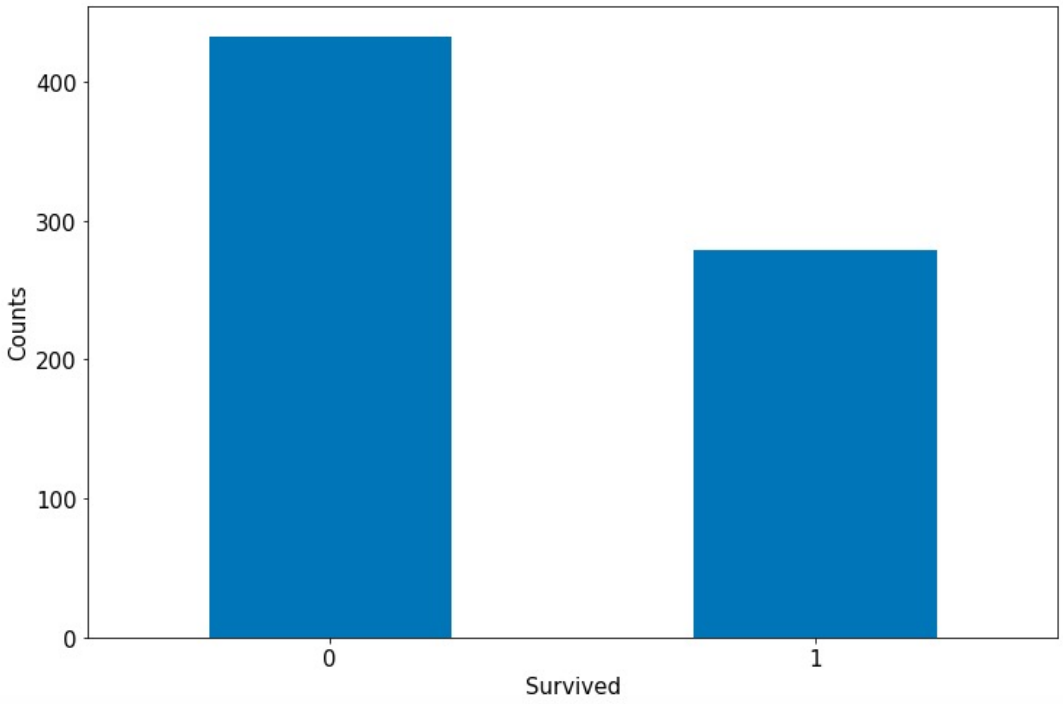
- Survived:0代表死亡，1代表存活【y标签】
- Pclass:乘客所持票类，有三种值(1,2,3) 【转换成onehot编码】
- Name:乘客姓名 【舍去】
- Sex:乘客性别 【转换成bool特征】

- Age:乘客年龄(有缺失) 【数值特征，添加“年龄是否缺失”作为辅助特征】
- SibSp:乘客兄弟姐妹/配偶的个数(整数值) 【数值特征】
- Parch:乘客父母/孩子的个数(整数值)【数值特征】
- Ticket:票号(字符串)【舍去】
- Fare:乘客所持票的价格(浮点数，0-500不等) 【数值特征】
- Cabin:乘客所在船舱(有缺失) 【添加“所在船舱是否缺失”作为辅助特征】
- Embarked:乘客登船港口:S、C、Q(有缺失)【转换成onehot编码，四维度 S,C,Q,nan】

利用Pandas的数据可视化功能我们可以简单地进行探索性数据分析EDA（Exploratory Data Analysis）。

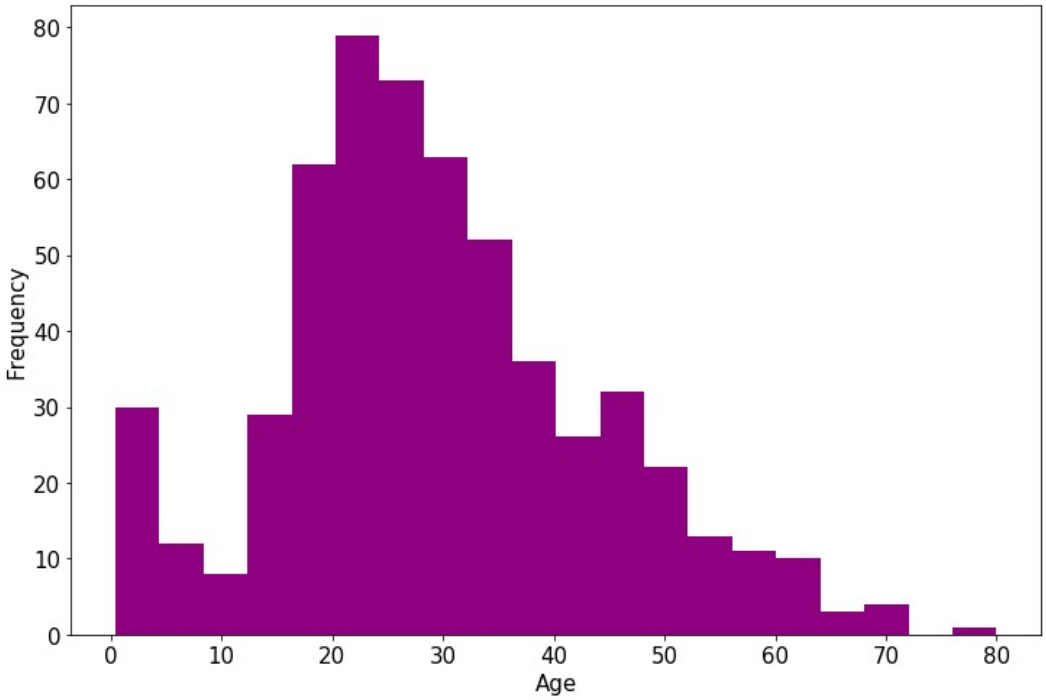
label分布情况

```
%matplotlib inline
%config InlineBackend.figure_format = 'png'
ax = dftrain_raw['Survived'].value_counts().plot(kind = 'bar',
     figsize = (12,8),fontsize=15,rot = 0)
ax.set_ylabel('Counts',fontsize = 15)
ax.set_xlabel('Survived',fontsize = 15)
plt.show()
```



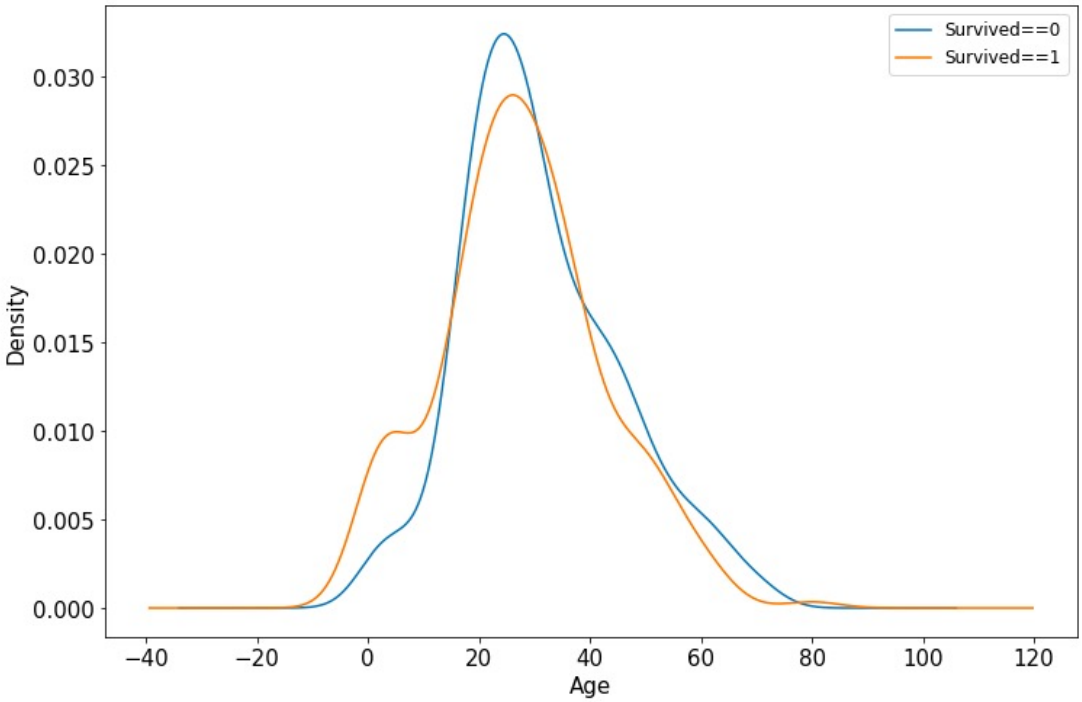
年龄分布情况

```
%matplotlib inline
%config InlineBackend.figure_format = 'png'
ax = dftrain_raw['Age'].plot(kind = 'hist',bins = 20,color= 'purple',
                    figsize = (12,8),fontsize=15)

ax.set_ylabel('Frequency',fontsize = 15)
ax.set_xlabel('Age',fontsize = 15)
plt.show()
```



年龄和label的相关性

```
%matplotlib inline
%config InlineBackend.figure_format = 'png'
ax = dftrain_raw.query('Survived == 0')['Age'].plot(kind = 'density',
                      figsize = (12,8),fontsize=15)
dftrain_raw.query('Survived == 1')['Age'].plot(kind = 'density',
                      figsize = (12,8),fontsize=15)
ax.legend(['Survived==0','Survived==1'],fontsize = 12)
ax.set_ylabel('Density',fontsize = 15)
ax.set_xlabel('Age',fontsize = 15)
plt.show()
```



下面为正式的数据预处理

```
def preprocessing(dfdata):

    dfresult= pd.DataFrame()

    #Pclass
    dfPclass = pd.get_dummies(dfdata['Pclass'])
    dfPclass.columns = ['Pclass_' +str(x) for x in dfPclass.columns ]
    dfresult = pd.concat([dfresult,dfPclass],axis = 1)

    #Sex
    dfSex = pd.get_dummies(dfdata['Sex'])
    dfresult = pd.concat([dfresult,dfSex],axis = 1)

    #Age
    dfresult['Age'] = dfdata['Age'].fillna(0)
    dfresult['Age_null'] = pd.isna(dfdata['Age']).astype('int32')

    #SibSp,Parch,Fare
    dfresult['SibSp'] = dfdata['SibSp']
    dfresult['Parch'] = dfdata['Parch']
    dfresult['Fare'] = dfdata['Fare']

    #Carbin
    dfresult['Cabin_null'] =  pd.isna(dfdata['Cabin']).astype('int32')

    #Embarked
    dfEmbarked = pd.get_dummies(dfdata['Embarked'],dummy_na=True)
    dfEmbarked.columns = ['Embarked_' + str(x) for x in dfEmbarked.columns]
    dfresult = pd.concat([dfresult,dfEmbarked],axis = 1)

    return(dfresult)

x_train = preprocessing(dftrain_raw).values
y_train = dftrain_raw[['Survived']].values

x_test = preprocessing(dftest_raw).values
y_test = dftest_raw[['Survived']].values

print("x_train.shape =", x_train.shape )
print("x_test.shape =", x_test.shape )

print("y_train.shape =", y_train.shape )
print("y_test.shape =", y_test.shape )
```

```
x_train.shape = (712, 15)
x_test.shape = (179, 15)
y_train.shape = (712, 1)
y_test.shape = (179, 1)
```

进一步使用DataLoader和TensorDataset封装成可以迭代的数据管道。

```
dl_train = DataLoader(TensorDataset(torch.tensor(x_train).float(),torch.tensor(y_train).float()),
                      shuffle = True, batch_size = 8)
dl_valid = DataLoader(TensorDataset(torch.tensor(x_test).float(),torch.tensor(y_test).float()),
                      shuffle = False, batch_size = 8)


# 测试数据管道
for features,labels in dl_train:
    print(features,labels)
    break
```

```
tensor([[  0.0000,   0.0000,   1.0000,   0.0000,   1.0000,   0.0000,   1.0000,
           0.0000,   0.0000,   7.8958,   1.0000,   0.0000,   0.0000,   1.0000,
           0.0000],
        [  1.0000,   0.0000,   0.0000,   0.0000,   1.0000,   0.0000,   1.0000,
           0.0000,   0.0000,  30.5000,   0.0000,   0.0000,   0.0000,   1.0000,
           0.0000],
        [  1.0000,   0.0000,   0.0000,   1.0000,   0.0000,  31.0000,   0.0000,
           1.0000,   0.0000, 113.2750,   0.0000,   1.0000,   0.0000,   0.0000,
           0.0000],
        [  1.0000,   0.0000,   0.0000,   0.0000,   1.0000,  60.0000,   0.0000,
           0.0000,   0.0000,  26.5500,   1.0000,   0.0000,   0.0000,   1.0000,
           0.0000],
        [  0.0000,   0.0000,   1.0000,   0.0000,   1.0000,  28.0000,   0.0000,
           0.0000,   0.0000,  22.5250,   1.0000,   0.0000,   0.0000,   1.0000,
           0.0000],
        [  0.0000,   0.0000,   1.0000,   0.0000,   1.0000,  32.0000,   0.0000,
           0.0000,   0.0000,   8.3625,   1.0000,   0.0000,   0.0000,   1.0000,
           0.0000],
        [  0.0000,   1.0000,   0.0000,   1.0000,   0.0000,  28.0000,   0.0000,
           0.0000,   0.0000,  13.0000,   1.0000,   0.0000,   0.0000,   1.0000,
           0.0000],
        [  1.0000,   0.0000,   0.0000,   0.0000,   1.0000,  36.0000,   0.0000,
           0.0000,   1.0000, 512.3292,   0.0000,   1.0000,   0.0000,   0.0000,
           0.0000]]) tensor([[0.],
        [1.],
        [1.],
        [0.],
        [0.],
        [0.],
        [1.],
        [1.]])
```

二，定义模型

使用Pytorch通常有三种方式构建模型：使用nn.Sequential按层顺序构建模型，继承nn.Module基类构建自定义模型，继承nn.Module基类构建模型并辅助应用模型容器进行封装。

此处选择使用最简单的nn.Sequential，按层顺序模型。

```
def create_net():
    net = nn.Sequential()
    net.add_module("linear1",nn.Linear(15,20))
    net.add_module("relu1",nn.ReLU())
    net.add_module("linear2",nn.Linear(20,15))
    net.add_module("relu2",nn.ReLU())
    net.add_module("linear3",nn.Linear(15,1))
    net.add_module("sigmoid",nn.Sigmoid())
    return net

net = create_net()
print(net)
```

```
Sequential(
  (linear1): Linear(in_features=15, out_features=20, bias=True)
  (relu1): ReLU()
  (linear2): Linear(in_features=20, out_features=15, bias=True)
  (relu2): ReLU()
  (linear3): Linear(in_features=15, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)
```

```
from torchkeras import summary
summary(net,input_shape=(15,))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1                   [-1, 20]             320
              ReLU-2                   [-1, 20]               0
            Linear-3                   [-1, 15]             315
              ReLU-4                   [-1, 15]               0
            Linear-5                    [-1, 1]              16
           Sigmoid-6                    [-1, 1]               0
================================================================
Total params: 651
Trainable params: 651
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.000057
Forward/backward pass size (MB): 0.000549
Params size (MB): 0.002483
Estimated Total Size (MB): 0.003090
----------------------------------------------------------------
```

三，训练模型

Pytorch通常需要用户编写自定义训练循环，训练循环的代码风格因人而异。

有3类典型的训练循环代码风格：脚本形式训练循环，函数形式训练循环，类形式训练循环。

此处介绍一种较通用的脚本形式。

```
from sklearn.metrics import accuracy_score

loss_func = nn.BCELoss()
optimizer = torch.optim.Adam(params=net.parameters(),lr = 0.01)
metric_func = lambda y_pred,y_true: accuracy_score(y_true.data.numpy(),y_pred.data.numpy()>0.5)
metric_name = "accuracy"
```

```
epochs = 10
log_step_freq = 30

dfhistory = pd.DataFrame(columns = ["epoch","loss",metric_name,"val_loss","val_"+metric_name])
print("Start Training...")
nowtime = datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S')
print("==========="*8 + "%s"%nowtime)

for epoch in range(1,epochs+1):

    # 1，训练循环-------------------------------------------------
    net.train()
    loss_sum = 0.0
    metric_sum = 0.0
    step = 1

    for step, (features,labels) in enumerate(dl_train, 1):

        # 梯度清零
        optimizer.zero_grad()

        # 正向传播求损失
        predictions = net(features)
        loss = loss_func(predictions,labels)
        metric = metric_func(predictions,labels)

        # 反向传播求梯度
        loss.backward()
        optimizer.step()

        # 打印batch级别日志
        loss_sum += loss.item()
        metric_sum += metric.item()
        if step%log_step_freq == 0:
            print(("[step = %d] loss: %.3f, "+metric_name+": %.3f") %
                  (step, loss_sum/step, metric_sum/step))

    # 2，验证循环-------------------------------------------------
    net.eval()
    val_loss_sum = 0.0
    val_metric_sum = 0.0
    val_step = 1

    for val_step, (features,labels) in enumerate(dl_valid, 1):
        # 关闭梯度计算
        with torch.no_grad():
            predictions = net(features)
            val_loss = loss_func(predictions,labels)
            val_metric = metric_func(predictions,labels)
        val_loss_sum += val_loss.item()
        val_metric_sum += val_metric.item()
```

```
    # 3，记录日志-----------------------------------------------------
    info = (epoch, loss_sum/step, metric_sum/step,
            val_loss_sum/val_step, val_metric_sum/val_step)
    dfhistory.loc[epoch-1] = info

    # 打印epoch级别日志
    print(("\nEPOCH = %d, loss = %.3f,"+ metric_name + \
          "  = %.3f, val_loss = %.3f, "+"val_"+ metric_name+" = %.3f")
          %info)
    nowtime = datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S')
    print("\n"+"=========="*8 + "%s"%nowtime)

print('Finished Training...')
```

```
Start Training...
==========================================================================2020-06-17 20:53:49
[step = 30] loss: 0.703, accuracy: 0.583
[step = 60] loss: 0.629, accuracy: 0.675

EPOCH = 1, loss = 0.643,accuracy  = 0.673, val_loss = 0.621, val_accuracy = 0.725

==========================================================================2020-06-17 20:53:49
[step = 30] loss: 0.653, accuracy: 0.662
[step = 60] loss: 0.624, accuracy: 0.673

EPOCH = 2, loss = 0.621,accuracy  = 0.669, val_loss = 0.519, val_accuracy = 0.708

==========================================================================2020-06-17 20:53:49
[step = 30] loss: 0.582, accuracy: 0.688
[step = 60] loss: 0.555, accuracy: 0.723

EPOCH = 3, loss = 0.543,accuracy  = 0.740, val_loss = 0.516, val_accuracy = 0.741

==========================================================================2020-06-17 20:53:49
[step = 30] loss: 0.563, accuracy: 0.721
[step = 60] loss: 0.528, accuracy: 0.752

EPOCH = 4, loss = 0.515,accuracy  = 0.764, val_loss = 0.471, val_accuracy = 0.777

==========================================================================2020-06-17 20:53:50
[step = 30] loss: 0.433, accuracy: 0.783
[step = 60] loss: 0.477, accuracy: 0.785

EPOCH = 5, loss = 0.489,accuracy  = 0.785, val_loss = 0.447, val_accuracy = 0.804

==========================================================================2020-06-17 20:53:50
[step = 30] loss: 0.460, accuracy: 0.812
[step = 60] loss: 0.477, accuracy: 0.798

EPOCH = 6, loss = 0.474,accuracy  = 0.798, val_loss = 0.451, val_accuracy = 0.772

==========================================================================2020-06-17 20:53:50
[step = 30] loss: 0.516, accuracy: 0.792
[step = 60] loss: 0.496, accuracy: 0.779

EPOCH = 7, loss = 0.473,accuracy  = 0.794, val_loss = 0.485, val_accuracy = 0.783

==========================================================================2020-06-17 20:53:50
[step = 30] loss: 0.472, accuracy: 0.779
[step = 60] loss: 0.487, accuracy: 0.794

EPOCH = 8, loss = 0.474,accuracy  = 0.791, val_loss = 0.446, val_accuracy = 0.788

==========================================================================2020-06-17 20:53:50
[step = 30] loss: 0.492, accuracy: 0.771
```

```
[step = 60] loss: 0.445, accuracy: 0.800

EPOCH = 9, loss = 0.464,accuracy  = 0.796, val_loss = 0.519, val_accuracy = 0.746

==========================================================================================2020-06-17 20:53:50
[step = 30] loss: 0.436, accuracy: 0.796
[step = 60] loss: 0.460, accuracy: 0.794

EPOCH = 10, loss = 0.462,accuracy  = 0.787, val_loss = 0.415, val_accuracy = 0.810

==========================================================================================2020-06-17 20:53:51
Finished Training...
```

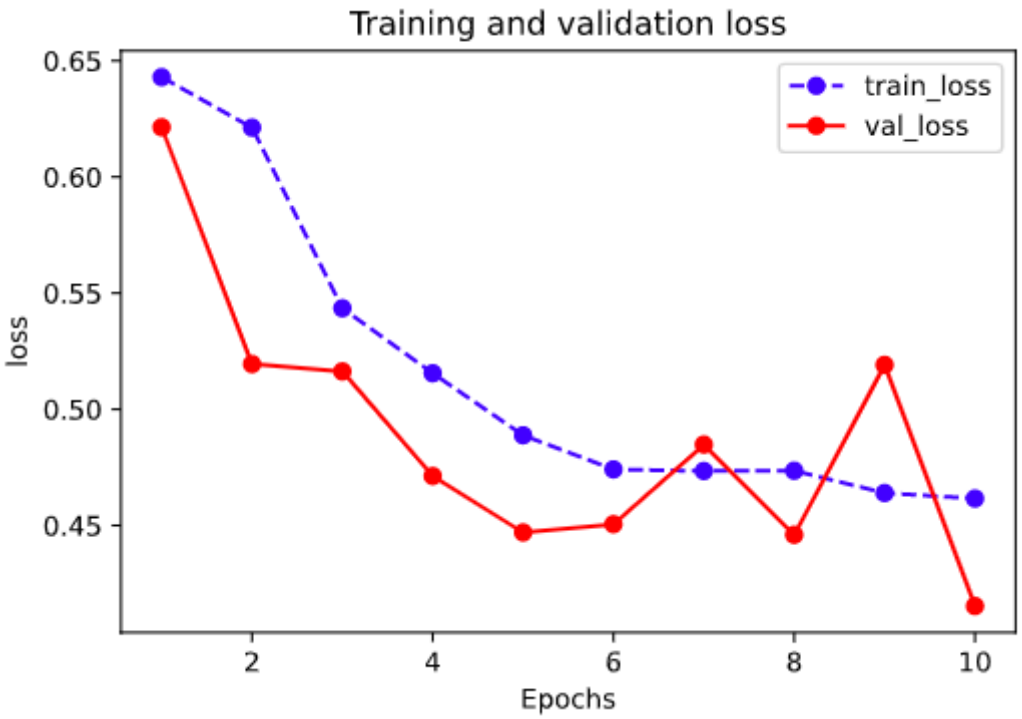
四，评估模型

我们首先评估一下模型在训练集和验证集上的效果。

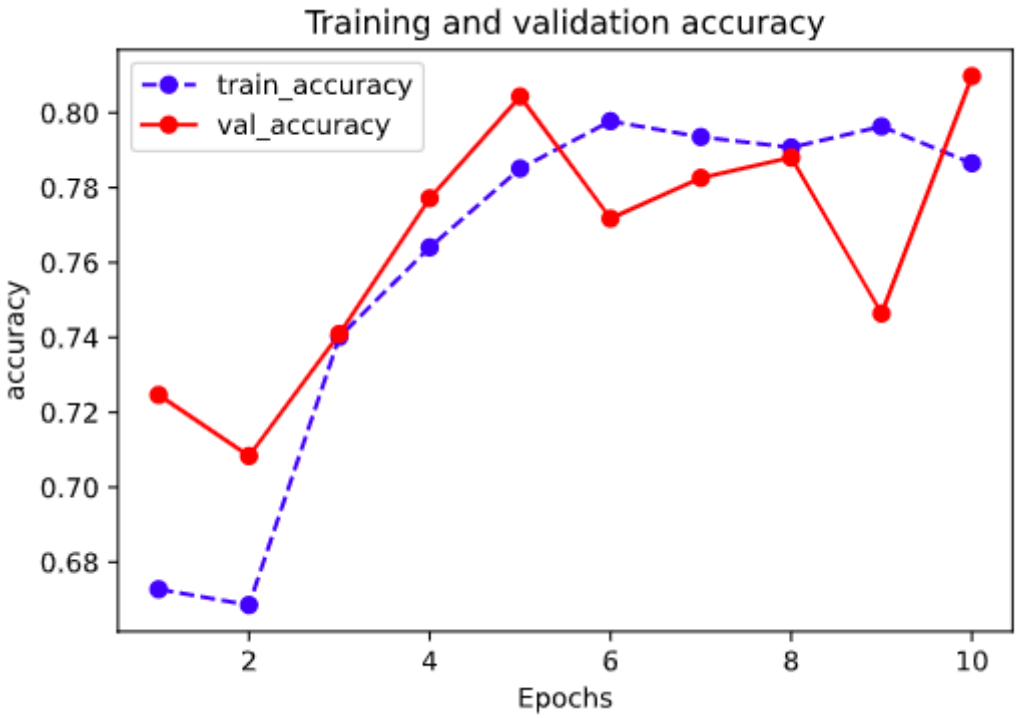
```
dfhistory
```

	epoch	loss	accuracy	val_loss	val_accuracy
0	1.0	0.642938	0.672753	0.621374	0.724638
1	2.0	0.621222	0.668539	0.519458	0.708333
2	3.0	0.543472	0.740169	0.516288	0.740942
3	4.0	0.515500	0.764045	0.471298	0.777174
4	5.0	0.488813	0.785112	0.446989	0.804348
5	6.0	0.474139	0.797753	0.450545	0.771739
6	7.0	0.473447	0.793539	0.484819	0.782609
7	8.0	0.473565	0.790730	0.445983	0.788043
8	9.0	0.463975	0.796348	0.519101	0.746377
9	10.0	0.461599	0.786517	0.415316	0.809783

```
%matplotlib inline
%config InlineBackend.figure_format = 'svg'

import matplotlib.pyplot as plt

def plot_metric(dfhistory, metric):
    train_metrics = dfhistory[metric]
    val_metrics = dfhistory['val_'+metric]
    epochs = range(1, len(train_metrics) + 1)
    plt.plot(epochs, train_metrics, 'bo--')
    plt.plot(epochs, val_metrics, 'ro-')
    plt.title('Training and validation '+ metric)
    plt.xlabel("Epochs")
    plt.ylabel(metric)
    plt.legend(["train_"+metric, 'val_'+metric])
    plt.show()


plot_metric(dfhistory,"loss")
```




```
plot_metric(dfhistory,"accuracy")
```



五，使用模型

```
#预测概率
y_pred_probs = net(torch.tensor(x_test[0:10]).float()).data
y_pred_probs
```

```
tensor([[0.0119],
        [0.6029],
        [0.2970],
        [0.5717],
        [0.5034],
        [0.8655],
        [0.0572],
        [0.9182],
        [0.5038],
        [0.1739]])
```

```
#预测类别
y_pred = torch.where(y_pred_probs>0.5,
        torch.ones_like(y_pred_probs),torch.zeros_like(y_pred_probs))
y_pred
```

```
tensor([[0.],
        [1.],
        [0.],
        [1.],
        [1.],
        [1.],
        [0.],
        [1.],
        [1.],
        [0.]])
```

六，保存模型

Pytorch 有两种保存模型的方式，都是通过调用pickle序列化方法实现的。

第一种方法只保存模型参数。

第二种方法保存完整模型。

推荐使用第一种，第二种方法可能在切换设备和目录的时候出现各种问题。

1，保存模型参数(推荐)

```
print(net.state_dict().keys())
```

```
odict_keys(['linear1.weight', 'linear1.bias', 'linear2.weight', 'linear2.bias', 'linear3.weight', 'linear3.bias'])
```

```
# 保存模型参数

torch.save(net.state_dict(), "./data/net_parameter.pkl")

net_clone = create_net()
net_clone.load_state_dict(torch.load("./data/net_parameter.pkl"))

net_clone.forward(torch.tensor(x_test[0:10]).float()).data
```

```
tensor([[0.0119],
        [0.6029],
        [0.2970],
        [0.5717],
        [0.5034],
        [0.8655],
        [0.0572],
        [0.9182],
        [0.5038],
        [0.1739]])
```

2，保存完整模型(不推荐)

```
torch.save(net, './data/net_model.pkl')
net_loaded = torch.load('./data/net_model.pkl')
net_loaded(torch.tensor(x_test[0:10]).float()).data
```

```
tensor([[0.0119],
        [0.6029],
        [0.2970],
        [0.5717],
        [0.5034],
        [0.8655],
        [0.0572],
        [0.9182],
        [0.5038],
        [0.1739]])
```

如果对本书内容理解上有需要进一步和作者交流的地方，欢迎在公众号"Python与算法之美"下留言。作者时间和精力有限，会酌情予以回复。

也可以在公众号后台回复关键字：**加群**，加入读者交流群和大家讨论。

1-2,图片数据建模流程范例

```
import os
import datetime

#打印时间
def printbar():
    nowtime = datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S')
    print("\n"+"=========="*8 + "%s"%nowtime)

#mac系统上pytorch和matplotlib在jupyter中同时跑需要更改环境变量
os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"
```

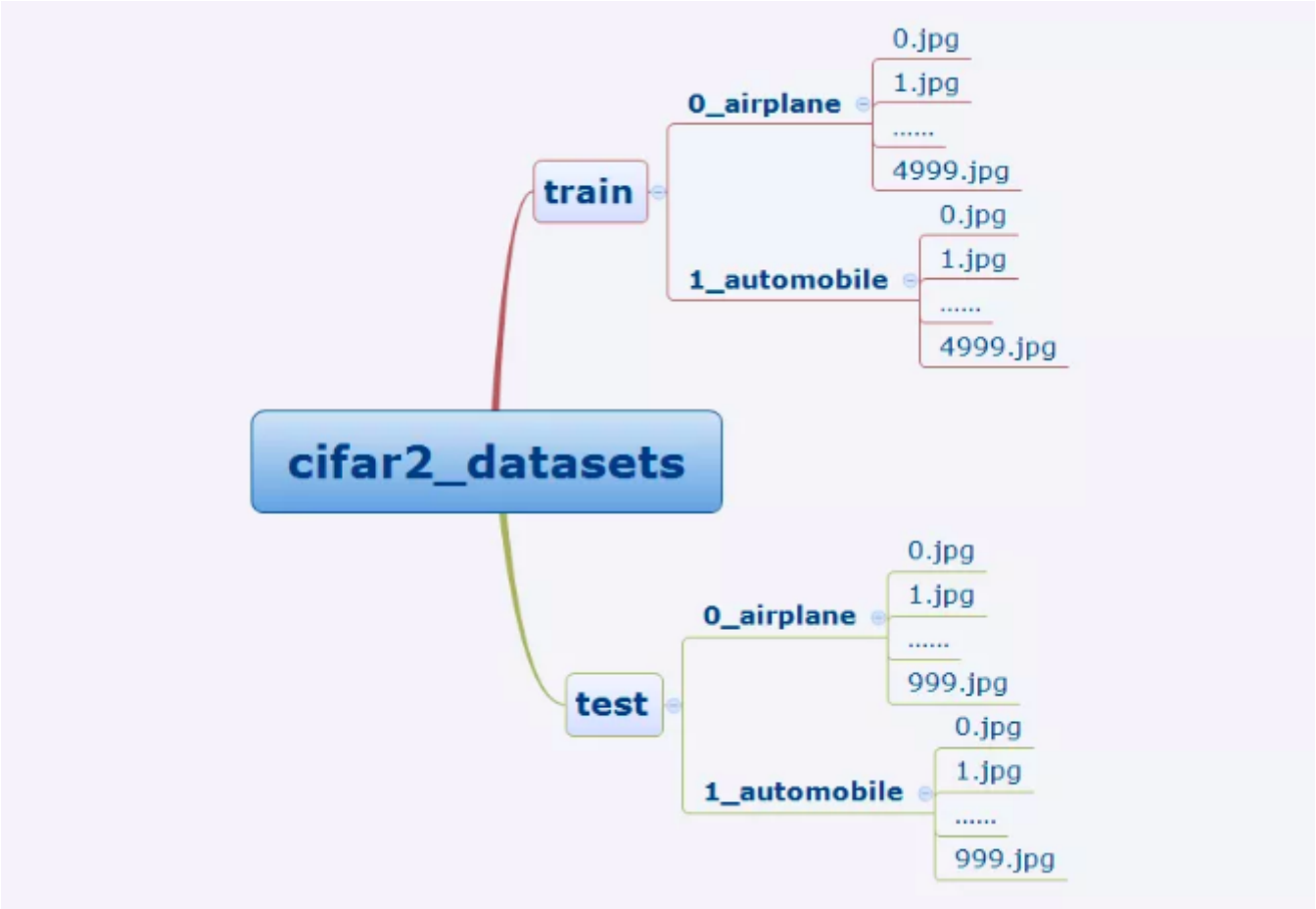
一，准备数据

cifar2数据集为cifar10数据集的子集，只包括前两种类别airplane和automobile。

训练集有airplane和automobile图片各5000张，测试集有airplane和automobile图片各1000张。

cifar2任务的目标是训练一个模型来对飞机airplane和机动车automobile两种图片进行分类。

我们准备的Cifar2数据集的文件结构如下所示。



在Pytorch中构建图片数据管道通常有三种方法。

第一种是使用 torchvision中的datasets.ImageFolder来读取图片然后用 DataLoader来并行加载。

第二种是通过继承 torch.utils.data.Dataset 实现用户自定义读取逻辑然后用 DataLoader来并行加载。

第三种方法是读取用户自定义数据集的通用方法，既可以读取图片数据集，也可以读取文本数据集。

本篇我们介绍第一种方法。

```
import torch
from torch import nn
from torch.utils.data import Dataset,DataLoader
from torchvision import transforms,datasets


transform_train = transforms.Compose(
    [transforms.ToTensor()])
transform_valid = transforms.Compose(
    [transforms.ToTensor()])
```

```
ds_train = datasets.ImageFolder("./data/cifar2/train/",
            transform = transform_train,target_transform= lambda t:torch.tensor([t]).float())
ds_valid = datasets.ImageFolder("./data/cifar2/test/",
            transform = transform_valid,target_transform= lambda t:torch.tensor([t]).float())

print(ds_train.class_to_idx)
```

```
{'0_airplane': 0, '1_automobile': 1}
```

```
dl_train = DataLoader(ds_train,batch_size = 50,shuffle = True,num_workers=3)
dl_valid = DataLoader(ds_valid,batch_size = 50,shuffle = True,num_workers=3)
```

```
%matplotlib inline
%config InlineBackend.figure_format = 'svg'

#查看部分样本
from matplotlib import pyplot as plt

plt.figure(figsize=(8,8))
for i in range(9):
    img,label = ds_train[i]
    img = img.permute(1,2,0)
    ax=plt.subplot(3,3,i+1)
    ax.imshow(img.numpy())
    ax.set_title("label = %d"%label.item())
    ax.set_xticks([])
    ax.set_yticks([])
plt.show()
```


```
# Pytorch的图片默认顺序是 Batch,Channel,Width,Height
for x,y in dl_train:
    print(x.shape,y.shape)
    break
```

```
torch.Size([50, 3, 32, 32]) torch.Size([50, 1])
```

二，定义模型

使用Pytorch通常有三种方式构建模型：使用nn.Sequential按层顺序构建模型，继承nn.Module基类构建自定义模型，继承nn.Module基类构建模型并辅助应用模型容器

(nn.Sequential,nn.ModuleList,nn.ModuleDict)进行封装。

此处选择通过继承nn.Module基类构建自定义模型。

```
#测试AdaptiveMaxPool2d的效果
pool = nn.AdaptiveMaxPool2d((1,1))
t = torch.randn(10,8,32,32)
pool(t).shape
```

```
torch.Size([10, 8, 1, 1])
```

```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=3,out_channels=32,kernel_size = 3)
        self.pool = nn.MaxPool2d(kernel_size = 2,stride = 2)
        self.conv2 = nn.Conv2d(in_channels=32,out_channels=64,kernel_size = 5)
        self.dropout = nn.Dropout2d(p = 0.1)
        self.adaptive_pool = nn.AdaptiveMaxPool2d((1,1))
        self.flatten = nn.Flatten()
        self.linear1 = nn.Linear(64,32)
        self.relu = nn.ReLU()
        self.linear2 = nn.Linear(32,1)
        self.sigmoid = nn.Sigmoid()

    def forward(self,x):
        x = self.conv1(x)
        x = self.pool(x)
        x = self.conv2(x)
        x = self.pool(x)
        x = self.dropout(x)
        x = self.adaptive_pool(x)
        x = self.flatten(x)
        x = self.linear1(x)
        x = self.relu(x)
        x = self.linear2(x)
        y = self.sigmoid(x)
        return y

net = Net()
print(net)
```

```
Net(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1))
  (dropout): Dropout2d(p=0.1, inplace=False)
  (adaptive_pool): AdaptiveMaxPool2d(output_size=(1, 1))
  (flatten): Flatten()
  (linear1): Linear(in_features=64, out_features=32, bias=True)
  (relu): ReLU()
  (linear2): Linear(in_features=32, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)
```

```
import torchkeras
torchkeras.summary(net,input_shape= (3,32,32))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 30, 30]             896
         MaxPool2d-2           [-1, 32, 15, 15]               0
            Conv2d-3           [-1, 64, 11, 11]          51,264
         MaxPool2d-4             [-1, 64, 5, 5]               0
         Dropout2d-5             [-1, 64, 5, 5]               0
 AdaptiveMaxPool2d-6             [-1, 64, 1, 1]               0
           Flatten-7                   [-1, 64]               0
            Linear-8                   [-1, 32]           2,080
              ReLU-9                   [-1, 32]               0
           Linear-10                    [-1, 1]              33
          Sigmoid-11                    [-1, 1]               0
================================================================
Total params: 54,273
Trainable params: 54,273
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.011719
Forward/backward pass size (MB): 0.359634
Params size (MB): 0.207035
Estimated Total Size (MB): 0.578388
----------------------------------------------------------------
```

三，训练模型

Pytorch通常需要用户编写自定义训练循环，训练循环的代码风格因人而异。

有3类典型的训练循环代码风格：脚本形式训练循环，函数形式训练循环，类形式训练循环。

此处介绍一种较通用的函数形式训练循环。

```
import pandas as pd
from sklearn.metrics import roc_auc_score

model = net
model.optimizer = torch.optim.SGD(model.parameters(),lr = 0.01)
model.loss_func = torch.nn.BCELoss()
model.metric_func = lambda y_pred,y_true: roc_auc_score(y_true.data.numpy(),y_pred.data.numpy())
model.metric_name = "auc"



def train_step(model,features,labels):

    # 训练模式，dropout层发生作用
    model.train()

    # 梯度清零
    model.optimizer.zero_grad()

    # 正向传播求损失
    predictions = model(features)
    loss = model.loss_func(predictions,labels)
    metric = model.metric_func(predictions,labels)

    # 反向传播求梯度
    loss.backward()
    model.optimizer.step()

    return loss.item(),metric.item()

def valid_step(model,features,labels):

    # 预测模式，dropout层不发生作用
    model.eval()
    # 关闭梯度计算
    with torch.no_grad():
        predictions = model(features)
        loss = model.loss_func(predictions,labels)
        metric = model.metric_func(predictions,labels)

    return loss.item(), metric.item()


# 测试train_step效果
features,labels = next(iter(dl_train))
train_step(model,features,labels)
```

(0.6922046542167664, 0.5088566827697262)

```
def train_model(model,epochs,dl_train,dl_valid,log_step_freq):

    metric_name = model.metric_name
    dfhistory = pd.DataFrame(columns = ["epoch","loss",metric_name,"val_loss","val_"+metric_name
    print("Start Training...")
    nowtime = datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S')
    print("=========="*8 + "%s"%nowtime)

    for epoch in range(1,epochs+1):

        # 1，训练循环-------------------------------------------------
        loss_sum = 0.0
        metric_sum = 0.0
        step = 1

        for step, (features,labels) in enumerate(dl_train, 1):

            loss,metric = train_step(model,features,labels)

            # 打印batch级别日志
            loss_sum += loss
            metric_sum += metric
            if step%log_step_freq == 0:
                print(("[step = %d] loss: %.3f, "+metric_name+": %.3f") %
                      (step, loss_sum/step, metric_sum/step))

        # 2，验证循环-------------------------------------------------
        val_loss_sum = 0.0
        val_metric_sum = 0.0
        val_step = 1

        for val_step, (features,labels) in enumerate(dl_valid, 1):

            val_loss,val_metric = valid_step(model,features,labels)

            val_loss_sum += val_loss
            val_metric_sum += val_metric

        # 3，记录日志-------------------------------------------------
        info = (epoch, loss_sum/step, metric_sum/step,
                val_loss_sum/val_step, val_metric_sum/val_step)
        dfhistory.loc[epoch-1] = info

        # 打印epoch级别日志
        print(("\nEPOCH = %d, loss = %.3f,"+ metric_name + \
              "  = %.3f, val_loss = %.3f, "+"val_"+ metric_name+" = %.3f")
              %info)
        nowtime = datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S')
        print("\n"+"=========="*8 + "%s"%nowtime)

    print('Finished Training...')
```

```
    return dfhistory

epochs = 20

dfhistory = train_model(model,epochs,dl_train,dl_valid,log_step_freq = 50)
```

```
Start Training...
================================================================================2020-06-28 20:47:56
[step = 50] loss: 0.691, auc: 0.627
[step = 100] loss: 0.690, auc: 0.673
[step = 150] loss: 0.688, auc: 0.699
[step = 200] loss: 0.686, auc: 0.716

EPOCH = 1, loss = 0.686,auc  = 0.716, val_loss = 0.678, val_auc = 0.806

================================================================================2020-06-28 20:48:18
[step = 50] loss: 0.677, auc: 0.780
[step = 100] loss: 0.675, auc: 0.775
[step = 150] loss: 0.672, auc: 0.782
[step = 200] loss: 0.669, auc: 0.779

EPOCH = 2, loss = 0.669,auc  = 0.779, val_loss = 0.651, val_auc = 0.815

......

================================================================================2020-06-28 20:54:24
[step = 50] loss: 0.386, auc: 0.914
[step = 100] loss: 0.392, auc: 0.913
[step = 150] loss: 0.395, auc: 0.911
[step = 200] loss: 0.398, auc: 0.911

EPOCH = 19, loss = 0.398,auc  = 0.911, val_loss = 0.449, val_auc = 0.924

================================================================================2020-06-28 20:54:43
[step = 50] loss: 0.416, auc: 0.917
[step = 100] loss: 0.417, auc: 0.916
[step = 150] loss: 0.404, auc: 0.918
[step = 200] loss: 0.402, auc: 0.918

EPOCH = 20, loss = 0.402,auc  = 0.918, val_loss = 0.535, val_auc = 0.925

================================================================================2020-06-28 20:55:03
Finished Training...
```

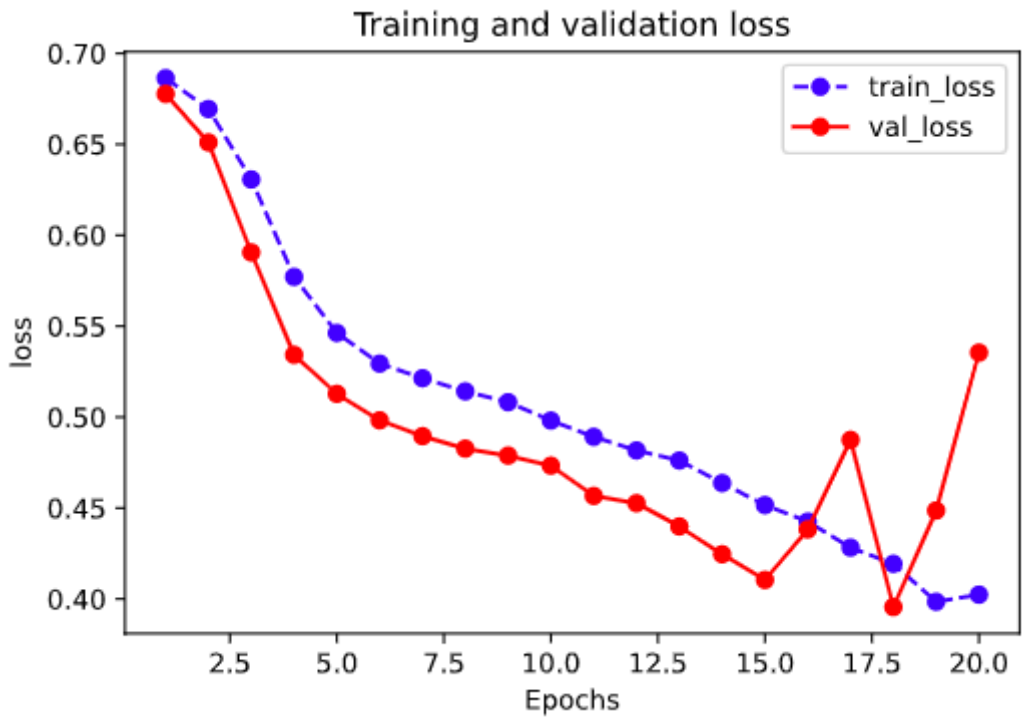
四，评估模型

dfhistory

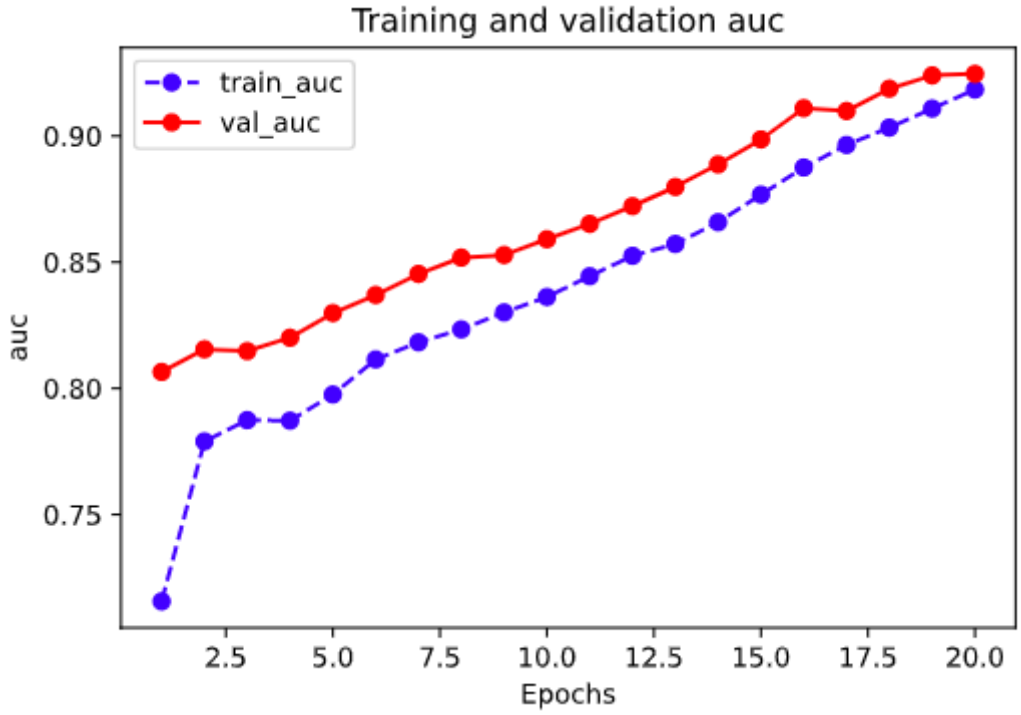
	epoch	loss	auc	val_loss	val_auc
0	1.0	0.686391	0.715593	0.677731	0.806496
1	2.0	0.669409	0.778992	0.651062	0.815481
2	3.0	0.630697	0.787418	0.590554	0.814759
3	4.0	0.577115	0.787207	0.534236	0.820052
4	5.0	0.546258	0.797564	0.512792	0.829819
5	6.0	0.529432	0.811458	0.498243	0.836977
6	7.0	0.521385	0.818241	0.489520	0.845349
7	8.0	0.514157	0.823361	0.482658	0.851799
8	9.0	0.508203	0.830094	0.478827	0.852824
9	10.0	0.498108	0.836203	0.473294	0.859092
10	11.0	0.489155	0.844427	0.456667	0.865150
11	12.0	0.481672	0.852523	0.452659	0.872201
12	13.0	0.476126	0.857189	0.439877	0.879786
13	14.0	0.463726	0.865842	0.424583	0.888724
14	15.0	0.451618	0.876747	0.410440	0.898626
15	16.0	0.442572	0.887507	0.438220	0.911008
16	17.0	0.428231	0.896443	0.487483	0.909889
17	18.0	0.419311	0.903312	0.395545	0.918737
18	19.0	0.398468	0.910851	0.448646	0.923978
19	20.0	0.402349	0.918417	0.535476	0.924582

```
%matplotlib inline
%config InlineBackend.figure_format = 'svg'

import matplotlib.pyplot as plt

def plot_metric(dfhistory, metric):
    train_metrics = dfhistory[metric]
    val_metrics = dfhistory['val_'+metric]
    epochs = range(1, len(train_metrics) + 1)
    plt.plot(epochs, train_metrics, 'bo--')
    plt.plot(epochs, val_metrics, 'ro-')
    plt.title('Training and validation '+ metric)
    plt.xlabel("Epochs")
    plt.ylabel(metric)
    plt.legend(["train_"+metric, 'val_'+metric])
    plt.show()
```

```
plot_metric(dfhistory,"loss")
```




```
plot_metric(dfhistory,"auc")
```



五，使用模型

```
def predict(model,dl):
    model.eval()
    with torch.no_grad():
        result = torch.cat([model.forward(t[0]) for t in dl])
    return(result.data)


#预测概率
y_pred_probs = predict(model,dl_valid)
y_pred_probs
```

```
tensor([[8.4032e-01],
        [1.0407e-02],
        [5.4146e-04],
        ...,
        [1.4471e-02],
        [1.7673e-02],
        [4.5081e-01]])
```

```
#预测类别
y_pred = torch.where(y_pred_probs>0.5,
         torch.ones_like(y_pred_probs),torch.zeros_like(y_pred_probs))
y_pred
```

```
tensor([[1.],
        [0.],
        [0.],
        ...,
        [0.],
        [0.],
        [0.]])
```

六，保存模型

推荐使用保存参数方式保存Pytorch模型。

```
print(model.state_dict().keys())
```

```
odict_keys(['conv1.weight', 'conv1.bias', 'conv2.weight', 'conv2.bias', 'linear1.weight', 'linear1.bias', 'linear
```

```
# 保存模型参数

torch.save(model.state_dict(), "./data/model_parameter.pkl")

net_clone = Net()
net_clone.load_state_dict(torch.load("./data/model_parameter.pkl"))

predict(net_clone,dl_valid)
```

```
tensor([[0.0204],
        [0.7692],
        [0.4967],
        ...,
        [0.6078],
        [0.7182],
        [0.8251]])
```

如果对本书内容理解上有需要进一步和作者交流的地方，欢迎在公众号"Python与算法之美"下留言。作者时间和精力有限，会酌情予以回复。

也可以在公众号后台回复关键字：**加群**，加入读者交流群和大家讨论。

2-1,张量数据结构

Pytorch的基本数据结构是张量Tensor。张量即多维数组。Pytorch的张量和numpy中的array很类似。

本节我们主要介绍张量的数据类型、张量的维度、张量的尺寸、张量和numpy数组等基本概念。

一，张量的数据类型

张量的数据类型和numpy.array基本一一对应，但是不支持str类型。

包括:

torch.float64(torch.double),

torch.float32(torch.float),

torch.float16,

torch.int64(torch.long),

torch.int32(torch.int),

torch.int16,

torch.int8,

torch.uint8,

torch.bool

一般神经网络建模使用的都是torch.float32类型。

```
import numpy as np
import torch

# 自动推断数据类型

i = torch.tensor(1);print(i,i.dtype)
x = torch.tensor(2.0);print(x,x.dtype)
b = torch.tensor(True);print(b,b.dtype)
```

```
tensor(1) torch.int64
tensor(2.) torch.float32
tensor(True) torch.bool
```

```
# 指定数据类型

i = torch.tensor(1,dtype = torch.int32);print(i,i.dtype)
x = torch.tensor(2.0,dtype = torch.double);print(x,x.dtype)
```

```
tensor(1, dtype=torch.int32) torch.int32
tensor(2., dtype=torch.float64) torch.float64
```

```
# 使用特定类型构造函数

i = torch.IntTensor(1);print(i,i.dtype)
x = torch.Tensor(np.array(2.0));print(x,x.dtype) #等价于torch.FloatTensor
b = torch.BoolTensor(np.array([1,0,2,0])); print(b,b.dtype)
```

```
tensor([5], dtype=torch.int32) torch.int32
tensor(2.) torch.float32
tensor([ True, False,  True, False]) torch.bool
```

```
# 不同类型进行转换

i = torch.tensor(1); print(i,i.dtype)
x = i.float(); print(x,x.dtype) #调用 float方法转换成浮点类型
y = i.type(torch.float); print(y,y.dtype) #使用type函数转换成浮点类型
z = i.type_as(x);print(z,z.dtype) #使用type_as方法转换成某个Tensor相同类型
```

```
tensor(1) torch.int64
tensor(1.) torch.float32
tensor(1.) torch.float32
tensor(1.) torch.float32
```

二，张量的维度

不同类型的数据可以用不同维度(dimension)的张量来表示。

标量为0维张量，向量为1维张量，矩阵为2维张量。

彩色图像有rgb三个通道，可以表示为3维张量。

视频还有时间维，可以表示为4维张量。

可以简单地总结为：有几层中括号，就是多少维的张量。

```
scalar = torch.tensor(True)
print(scalar)
print(scalar.dim())  # 标量，0维张量
```

```
tensor(True)
0
```

```
vector = torch.tensor([1.0,2.0,3.0,4.0]) #向量，1维张量
print(vector)
print(vector.dim())
```

```
tensor([1., 2., 3., 4.])
1
```

```
matrix = torch.tensor([[1.0,2.0],[3.0,4.0]]) #矩阵，2维张量
print(matrix)
print(matrix.dim())
```

```
matrix = torch.tensor([[1.0,2.0],[3.0,4.0]]) #矩阵，2维张量
print(matrix)
print(matrix.dim())
```

```
tensor3 = torch.tensor([[[1.0,2.0],[3.0,4.0]],[[5.0,6.0],[7.0,8.0]]])  # 3维张量
print(tensor3)
print(tensor3.dim())
```

```
tensor([[[1., 2.],
         [3., 4.]],

        [[5., 6.],
         [7., 8.]]])
3
```

```
tensor4 = torch.tensor([[[[1.0,1.0],[2.0,2.0]],[[3.0,3.0],[4.0,4.0]]],
                        [[[5.0,5.0],[6.0,6.0]],[[7.0,7.0],[8.0,8.0]]]])  # 4维张量
print(tensor4)
print(tensor4.dim())
```

```
tensor([[[[1., 1.],
          [2., 2.]],

         [[3., 3.],
          [4., 4.]]],


        [[[5., 5.],
          [6., 6.]],

         [[7., 7.],
          [8., 8.]]]])
4
```

三，张量的尺寸

可以使用 shape属性或者 size()方法查看张量在每一维的长度.

可以使用view方法改变张量的尺寸。

如果view方法改变尺寸失败，可以使用reshape方法.

```
scalar = torch.tensor(True)
print(scalar.size())
print(scalar.shape)
```

```
torch.Size([])
torch.Size([4])
```

```
vector = torch.tensor([1.0,2.0,3.0,4.0])
print(vector.size())
print(vector.shape)
```

```
torch.Size([4])
torch.Size([4])
```

```
matrix = torch.tensor([[1.0,2.0],[3.0,4.0]])
print(matrix.size())
```

```
torch.Size([2, 2])
```

```
# 使用view可以改变张量尺寸

vector = torch.arange(0,12)
print(vector)
print(vector.shape)

matrix34 = vector.view(3,4)
print(matrix34)
print(matrix34.shape)

matrix43 = vector.view(4,-1) #-1表示该位置长度由程序自动推断
print(matrix43)
print(matrix43.shape)
```

```
tensor([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
torch.Size([12])
tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]])
torch.Size([3, 4])
tensor([[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8],
        [ 9, 10, 11]])
torch.Size([4, 3])
```

```
# 有些操作会让张量存储结构扭曲，直接使用view会失败，可以用reshape方法

matrix26 = torch.arange(0,12).view(2,6)
print(matrix26)
print(matrix26.shape)

# 转置操作让张量存储结构扭曲
matrix62 = matrix26.t()
print(matrix62.is_contiguous())


# 直接使用view方法会失败，可以使用reshape方法
#matrix34 = matrix62.view(3,4) #error!
matrix34 = matrix62.reshape(3,4) #等价于matrix34 = matrix62.contiguous().view(3,4)
print(matrix34)
```

```
tensor([[ 0,  1,  2,  3,  4,  5],
        [ 6,  7,  8,  9, 10, 11]])
torch.Size([2, 6])
False
tensor([[ 0,  6,  1,  7],
        [ 2,  8,  3,  9],
        [ 4, 10,  5, 11]])
```

四，张量和numpy数组

可以用numpy方法从Tensor得到numpy数组，也可以用torch.from_numpy从numpy数组得到Tensor。

这两种方法关联的Tensor和numpy数组是共享数据内存的。

如果改变其中一个，另外一个的值也会发生改变。

如果有需要，可以用张量的clone方法拷贝张量，中断这种关联。

此外，还可以使用item方法从标量张量得到对应的Python数值。

使用tolist方法从张量得到对应的Python数值列表。

```
import numpy as np
import torch


#torch.from_numpy函数从numpy数组得到Tensor

arr = np.zeros(3)
tensor = torch.from_numpy(arr)
print("before add 1:")
print(arr)
print(tensor)

print("\nafter add 1:")
np.add(arr,1, out = arr) #给 arr增加1，tensor也随之改变
print(arr)
print(tensor)
```

```
before add 1:
[0. 0. 0.]
tensor([0., 0., 0.], dtype=torch.float64)

after add 1:
[1. 1. 1.]
tensor([1., 1., 1.], dtype=torch.float64)
```

```
# numpy方法从Tensor得到numpy数组

tensor = torch.zeros(3)
arr = tensor.numpy()
print("before add 1:")
print(tensor)
print(arr)

print("\nafter add 1:")

#使用带下划线的方法表示计算结果会返回给调用 张量
tensor.add_(1) #给 tensor增加1, arr也随之改变
#或:  torch.add(tensor,1,out = tensor)
print(tensor)
print(arr)
```

```
before add 1:
tensor([0., 0., 0.])
[0. 0. 0.]

after add 1:
tensor([1., 1., 1.])
[1. 1. 1.]
```

```
# 可以用clone() 方法拷贝张量，中断这种关联

tensor = torch.zeros(3)

#使用clone方法拷贝张量，拷贝后的张量和原始张量内存独立
arr = tensor.clone().numpy() # 也可以使用tensor.data.numpy()
print("before add 1:")
print(tensor)
print(arr)

print("\nafter add 1:")

#使用 带下划线的方法表示计算结果会返回给调用 张量
tensor.add_(1) #给 tensor增加1，arr不再随之改变
print(tensor)
print(arr)
```

```
before add 1:
tensor([0., 0., 0.])
[0. 0. 0.]

after add 1:
tensor([1., 1., 1.])
[0. 0. 0.]
```

```
# item方法和tolist方法可以将张量转换成Python数值和数值列表
scalar = torch.tensor(1.0)
s = scalar.item()
print(s)
print(type(s))

tensor = torch.rand(2,2)
t = tensor.tolist()
print(t)
print(type(t))
```

```
1.0
<class 'float'>
[[0.8211846351623535, 0.20020723342895508], [0.011571824550628662, 0.2906131148338318]]
<class 'list'>
```

如果对本书内容理解上有需要进一步和作者交流的地方，欢迎在公众号"Python与算法之美"下留言。作者时间和精力有限，会酌情予以回复。

也可以在公众号后台回复关键字：**加群**，加入读者交流群和大家讨论。

2-2,自动微分机制

神经网络通常依赖反向传播求梯度来更新网络参数，求梯度过程通常是一件非常复杂而容易出错的事情。

而深度学习框架可以帮助我们自动地完成这种求梯度运算。

Pytorch一般通过反向传播 backward 方法 实现这种求梯度计算。该方法求得的梯度将存在对应自变量张量的grad属性下。

除此之外，也能够调用torch.autograd.grad 函数来实现求梯度计算。

这就是Pytorch的自动微分机制。

一，利用backward方法求导数

backward 方法通常在一个标量张量上调用，该方法求得的梯度将存在对应自变量张量的grad属性下。

如果调用的张量非标量，则要传入一个和它同形状 的gradient参数张量。

相当于用该gradient参数张量与调用张量作向量点乘，得到的标量结果再反向传播。

1, 标量的反向传播

```
import numpy as np
import torch

# f(x) = a*x**2 + b*x + c的导数

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

y.backward()
dy_dx = x.grad
print(dy_dx)
```

```
tensor(-2.)
```

2, 非标量的反向传播

```
import numpy as np
import torch

# f(x) = a*x**2 + b*x + c

x = torch.tensor([[0.0,0.0],[1.0,2.0]],requires_grad = True) # x需要被求导
a = torch.tensor(1.0)
b = torch.tensor(-2.0)
c = torch.tensor(1.0)
y = a*torch.pow(x,2) + b*x + c

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

print("x:\n",x)
print("y:\n",y)
y.backward(gradient = gradient)
x_grad = x.grad
print("x_grad:\n",x_grad)
```

```
x:
 tensor([[0., 0.],
        [1., 2.]], requires_grad=True)
y:
 tensor([[1., 1.],
        [0., 1.]], grad_fn=<AddBackward0>)
x_grad:
 tensor([[-2., -2.],
        [ 0.,  2.]])
```

3, 非标量的反向传播可以用标量的反向传播实现

```
import numpy as np
import torch

# f(x) = a*x**2 + b*x + c

x = torch.tensor([[0.0,0.0],[1.0,2.0]],requires_grad = True) # x需要被求导
a = torch.tensor(1.0)
b = torch.tensor(-2.0)
c = torch.tensor(1.0)
y = a*torch.pow(x,2) + b*x + c

gradient = torch.tensor([[1.0,1.0],[1.0,1.0]])
z = torch.sum(y*gradient)

print("x:",x)
print("y:",y)
z.backward()
x_grad = x.grad
print("x_grad:\n",x_grad)
```

```
x: tensor([[0., 0.],
        [1., 2.]], requires_grad=True)
y: tensor([[1., 1.],
        [0., 1.]], grad_fn=<AddBackward0>)
x_grad:
 tensor([[-2., -2.],
        [ 0.,  2.]])
```

二，利用autograd.grad方法求导数

```python
import numpy as np
import torch

# f(x) = a*x**2 + b*x + c的导数

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


# create_graph 设置为 True 将允许创建更高阶的导数
dy_dx = torch.autograd.grad(y,x,create_graph=True)[0]
print(dy_dx.data)

# 求二阶导数
dy2_dx2 = torch.autograd.grad(dy_dx,x)[0]

print(dy2_dx2.data)
```

```
tensor(-2.)
tensor(2.)
```

```python
import numpy as np
import torch

x1 = torch.tensor(1.0,requires_grad = True) # x需要被求导
x2 = torch.tensor(2.0,requires_grad = True)

y1 = x1*x2
y2 = x1+x2


# 允许同时对多个自变量求导数
(dy1_dx1,dy1_dx2) = torch.autograd.grad(outputs=y1,inputs = [x1,x2],retain_graph = True)
print(dy1_dx1,dy1_dx2)

# 如果有多个因变量，相当于把多个因变量的梯度结果求和
(dy12_dx1,dy12_dx2) = torch.autograd.grad(outputs=[y1,y2],inputs = [x1,x2])
print(dy12_dx1,dy12_dx2)
```

```
tensor(2.) tensor(1.)
tensor(3.) tensor(2.)
```

三，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)


def f(x):
    result = a*torch.pow(x,2) + b*x + c
    return(result)

for i in range(500):
    optimizer.zero_grad()
    y = f(x)
    y.backward()
    optimizer.step()


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

```
y= tensor(0.) ; x= tensor(1.0000)
```

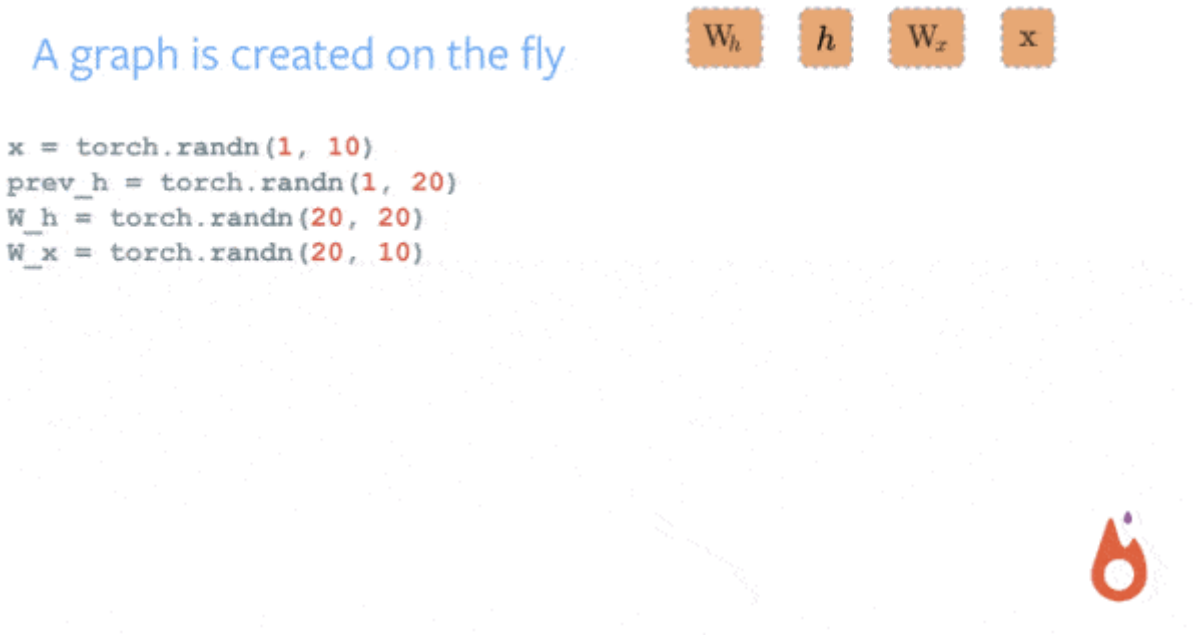
2-3,动态计算图

本节我们将介绍 Pytorch的动态计算图。

包括：

- 动态计算图简介
- 计算图中的Function
- 计算图和反向传播
- 叶子节点和非叶子节点
- 计算图在TensorBoard中的可视化

一，动态计算图简介



Pytorch的计算图由节点和边组成，节点表示张量或者Function，边表示张量和Function之间的依赖关系。

Pytorch中的计算图是动态图。这里的动态主要有两重含义。

第一层含义是：计算图的正向传播是立即执行的。无需等待完整的计算图创建完毕，每条语句都会在计算图中动态添加节点和边，并立即执行正向传播得到计算结果。

第二层含义是：计算图在反向传播后立即销毁。下次调用需要重新构建计算图。如果在程序中使用了backward方法执行了反向传播，或者利用torch.autograd.grad方法计算了梯度，那么创建的计算图会被立即销毁，释放存储空间，下次调用需要重新创建。

1，计算图的正向传播是立即执行的。

```
import torch
w = torch.tensor([[3.0,1.0]],requires_grad=True)
b = torch.tensor([[3.0]],requires_grad=True)
X = torch.randn(10,2)
Y = torch.randn(10,1)
Y_hat = X@w.t() + b  # Y_hat定义后其正向传播被立即执行，与其后面的loss创建语句无关
loss = torch.mean(torch.pow(Y_hat-Y,2))

print(loss.data)
print(Y_hat.data)
```

```
tensor(17.8969)
tensor([[3.2613],
        [4.7322],
        [4.5037],
        [7.5899],
        [7.0973],
        [1.3287],
        [6.1473],
        [1.3492],
        [1.3911],
        [1.2150]])
```

2，计算图在反向传播后立即销毁。

```
import torch
w = torch.tensor([[3.0,1.0]],requires_grad=True)
b = torch.tensor([[3.0]],requires_grad=True)
X = torch.randn(10,2)
Y = torch.randn(10,1)
Y_hat = X@w.t() + b  # Y_hat定义后其正向传播被立即执行，与其后面的loss创建语句无关
loss = torch.mean(torch.pow(Y_hat-Y,2))

#计算图在反向传播后立即销毁，如果需要保留计算图，需要设置retain_graph = True
loss.backward()  #loss.backward(retain_graph = True)

#loss.backward() #如果再次执行反向传播将报错
```

二，计算图中的Function

计算图中的 张量我们已经比较熟悉了，计算图中的另外一种节点是Function，实际上就是 Pytorch中各种对张量操作的函数。

这些Function和我们Python中的函数有一个较大的区别，那就是它同时包括正向计算逻辑和反向传播的逻辑。

我们可以通过继承torch.autograd.Function来创建这种支持反向传播的Function

```
class MyReLU(torch.autograd.Function):

    #正向传播逻辑，可以用ctx存储一些值，供反向传播使用。
    @staticmethod
    def forward(ctx, input):
        ctx.save_for_backward(input)
        return input.clamp(min=0)

    #反向传播逻辑
    @staticmethod
    def backward(ctx, grad_output):
        input, = ctx.saved_tensors
        grad_input = grad_output.clone()
        grad_input[input < 0] = 0
        return grad_input


import torch
w = torch.tensor([[3.0,1.0]],requires_grad=True)
b = torch.tensor([[3.0]],requires_grad=True)
X = torch.tensor([[-1.0,-1.0],[1.0,1.0]])
Y = torch.tensor([[2.0,3.0]])

relu = MyReLU.apply # relu现在也可以具有正向传播和反向传播功能
Y_hat = relu(X@w.t() + b)
loss = torch.mean(torch.pow(Y_hat-Y,2))

loss.backward()

print(w.grad)
print(b.grad)


tensor([[4.5000, 4.5000]])
tensor([[4.5000]])


# Y_hat的梯度函数即是我们自己所定义的 MyReLU.backward

print(Y_hat.grad_fn)


<torch.autograd.function.MyReLUBackward object at 0x1205a46c8>
```

三，计算图与反向传播

了解了Function的功能，我们可以简单地理解一下反向传播的原理和过程。理解该部分原理需要一些高等数学中求导链式法则的基础知识。

```
import torch

x = torch.tensor(3.0,requires_grad=True)
y1 = x + 1
y2 = 2*x
loss = (y1-y2)**2

loss.backward()
```

loss.backward()语句调用后，依次发生以下计算过程。

1，loss自己的grad梯度赋值为1，即对自身的梯度为1。

2，loss根据其自身梯度以及关联的backward方法，计算出其对应的自变量即y1和y2的梯度，将该值赋值到y1.grad和y2.grad。

3，y2和y1根据其自身梯度以及关联的backward方法, 分别计算出其对应的自变量x的梯度，x.grad将其收到的多个梯度值累加。

（注意，1,2,3步骤的求梯度顺序和对多个梯度值的累加规则恰好是求导链式法则的程序表述）

正因为求导链式法则衍生的梯度累加规则，张量的grad梯度不会自动清零，在需要的时候需要手动置零。

四，叶子节点和非叶子节点

执行下面代码，我们会发现 loss.grad并不是我们期望的1,而是 None。

类似地 y1.grad 以及 y2.grad也是 None.

这是为什么呢？这是由于它们不是叶子节点张量。

在反向传播过程中，只有 is_leaf=True 的叶子节点，需要求导的张量的导数结果才会被最后保留下来。

那么什么是叶子节点张量呢？叶子节点张量需要满足两个条件。

1，叶子节点张量是由用户直接创建的张量，而非由某个Function通过计算得到的张量。

2，叶子节点张量的 requires_grad属性必须为True.

Pytorch设计这样的规则主要是为了节约内存或者显存空间，因为几乎所有的时候，用户只会关心他自己直接创建的张量的梯度。

所有依赖于叶子节点张量的张量, 其requires_grad 属性必定是True的，但其梯度值只在计算过程中被用到，不会最终存储到grad属性中。

如果需要保留中间计算结果的梯度到grad属性中，可以使用 retain_grad方法。
如果仅仅是为了调试代码查看梯度值，可以利用register_hook打印日志。

```
import torch

x = torch.tensor(3.0,requires_grad=True)
y1 = x + 1
y2 = 2*x
loss = (y1-y2)**2

loss.backward()
print("loss.grad:", loss.grad)
print("y1.grad:", y1.grad)
print("y2.grad:", y2.grad)
print(x.grad)
```

```
loss.grad: None
y1.grad: None
y2.grad: None
tensor(4.)
```

```
print(x.is_leaf)
print(y1.is_leaf)
print(y2.is_leaf)
print(loss.is_leaf)
```

```
True
False
False
False
```

利用retain_grad可以保留非叶子节点的梯度值，利用register_hook可以查看非叶子节点的梯度值。

```
import torch

#正向传播
x = torch.tensor(3.0,requires_grad=True)
y1 = x + 1
y2 = 2*x
loss = (y1-y2)**2

#非叶子节点梯度显示控制
y1.register_hook(lambda grad: print('y1 grad: ', grad))
y2.register_hook(lambda grad: print('y2 grad: ', grad))
loss.retain_grad()

#反向传播
loss.backward()
print("loss.grad:", loss.grad)
print("x.grad:", x.grad)
```

```
y2 grad:  tensor(4.)
y1 grad:  tensor(-4.)
loss.grad: tensor(1.)
x.grad: tensor(4.)
```

五，计算图在TensorBoard中的可视化

可以利用 torch.utils.tensorboard 将计算图导出到 TensorBoard进行可视化。

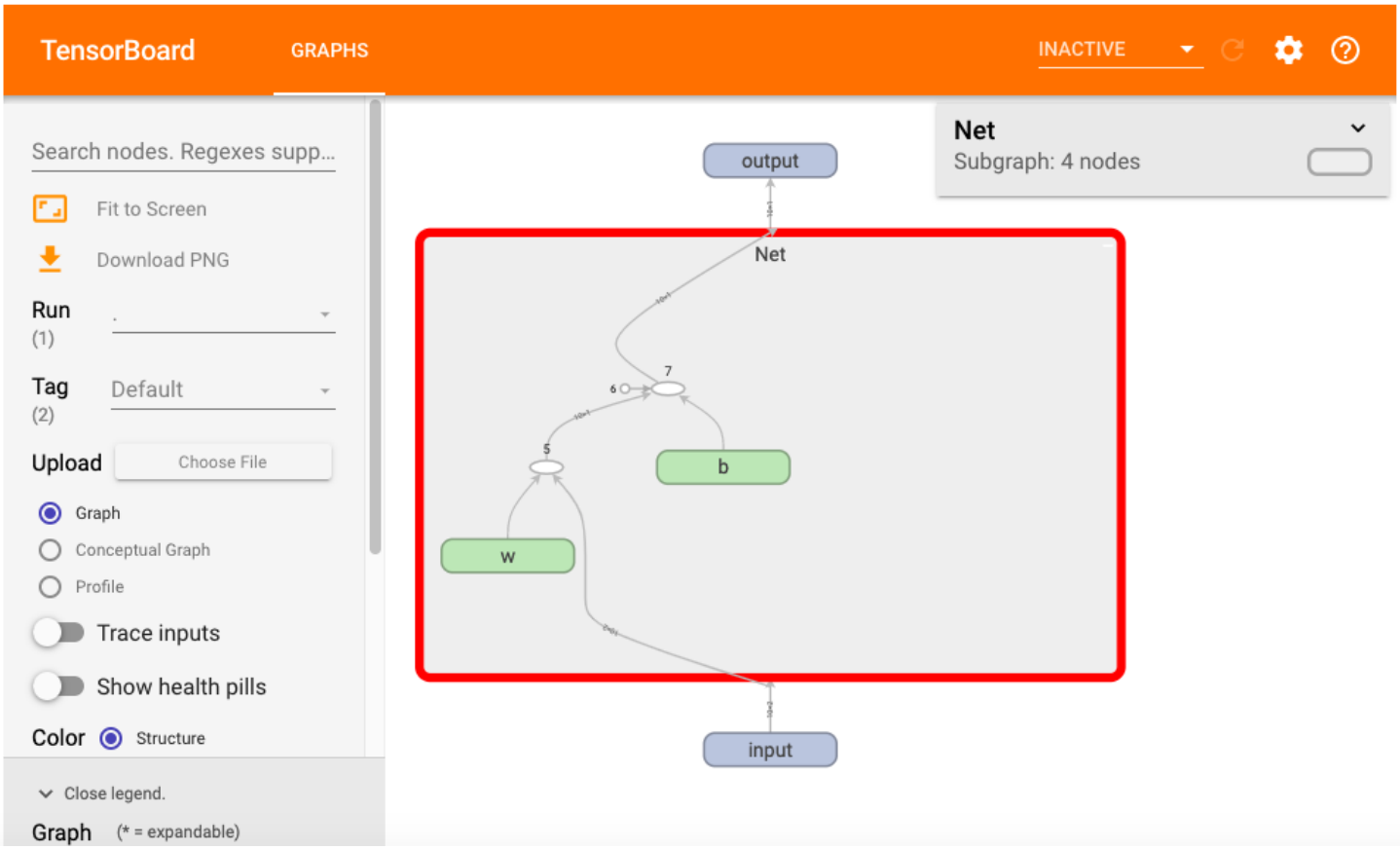
```
from torch import nn
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.w = nn.Parameter(torch.randn(2,1))
        self.b = nn.Parameter(torch.zeros(1,1))

    def forward(self, x):
        y = x@self.w + self.b
        return y

net = Net()
```

```
from torch.utils.tensorboard import SummaryWriter
writer = SummaryWriter('./data/tensorboard')
writer.add_graph(net,input_to_model = torch.rand(10,2))
writer.close()
```

```
%load_ext tensorboard
#%tensorboard --logdir ./data/tensorboard
```

```
from tensorboard import notebook
notebook.list()
```

```
#在tensorboard中查看模型
notebook.start("--logdir ./data/tensorboard")
```



3-1,低阶API示范

下面的范例使用Pytorch的低阶API实现线性回归模型和DNN二分类模型。

低阶API主要包括张量操作，计算图和自动微分。

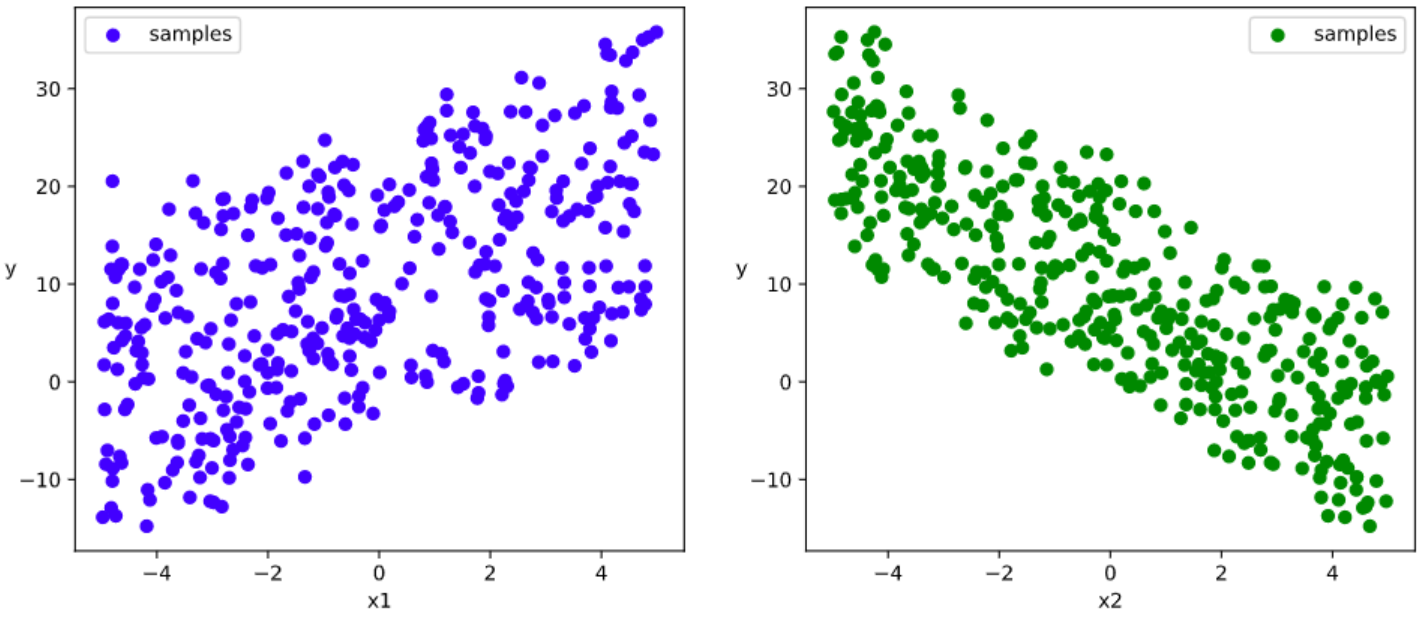
```
import os
import datetime

#打印时间
def printbar():
    nowtime = datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S')
    print("\n"+"=========="*8 + "%s"%nowtime)

#mac系统上pytorch和matplotlib在jupyter中同时跑需要更改环境变量
os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"
```

一，线性回归模型

1，准备数据

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
import torch
from torch import nn


#样本数量
n = 400

# 生成测试用数据集
X = 10*torch.rand([n,2])-5.0  #torch.rand是均匀分布
w0 = torch.tensor([[2.0],[-3.0]])
b0 = torch.tensor([[10.0]])
Y = X@w0 + b0 + torch.normal( 0.0,2.0,size = [n,1])  # @表示矩阵乘法,增加正态扰动
```

```
# 数据可视化

%matplotlib inline
%config InlineBackend.figure_format = 'svg'

plt.figure(figsize = (12,5))
ax1 = plt.subplot(121)
ax1.scatter(X[:,0].numpy(),Y[:,0].numpy(), c = "b",label = "samples")
ax1.legend()
plt.xlabel("x1")
plt.ylabel("y",rotation = 0)

ax2 = plt.subplot(122)
ax2.scatter(X[:,1].numpy(),Y[:,0].numpy(), c = "g",label = "samples")
ax2.legend()
plt.xlabel("x2")
plt.ylabel("y",rotation = 0)
plt.show()
```



```
# 构建数据管道迭代器
def data_iter(features, labels, batch_size=8):
    num_examples = len(features)
    indices = list(range(num_examples))
    np.random.shuffle(indices)  #样本的读取顺序是随机的
    for i in range(0, num_examples, batch_size):
        indexs = torch.LongTensor(indices[i: min(i + batch_size, num_examples)])
        yield  features.index_select(0, indexs), labels.index_select(0, indexs)

# 测试数据管道效果
batch_size = 8
(features,labels) = next(data_iter(X,Y,batch_size))
print(features)
print(labels)
```

```
tensor([[-4.3880,  1.3655],
        [-0.1082,  3.9533],
        [-2.6286,  2.7058],
        [ 1.0604, -1.8646],
        [-1.5805,  1.5406],
        [-2.6217, -3.2342],
        [ 2.3748, -0.6449],
        [-1.2478, -2.0509]])
tensor([[-0.2069],
        [-3.2494],
        [-6.9620],
        [17.0528],
        [ 1.1076],
        [17.2117],
        [16.1081],
        [14.7092]])
```

2，定义模型

```
# 定义模型
class LinearRegression:

    def __init__(self):
        self.w = torch.randn_like(w0,requires_grad=True)
        self.b = torch.zeros_like(b0,requires_grad=True)

    #正向传播
    def forward(self,x):
        return x@self.w + self.b

    # 损失函数
    def loss_func(self,y_pred,y_true):
        return torch.mean((y_pred - y_true)**2/2)

model = LinearRegression()
```

3，训练模型

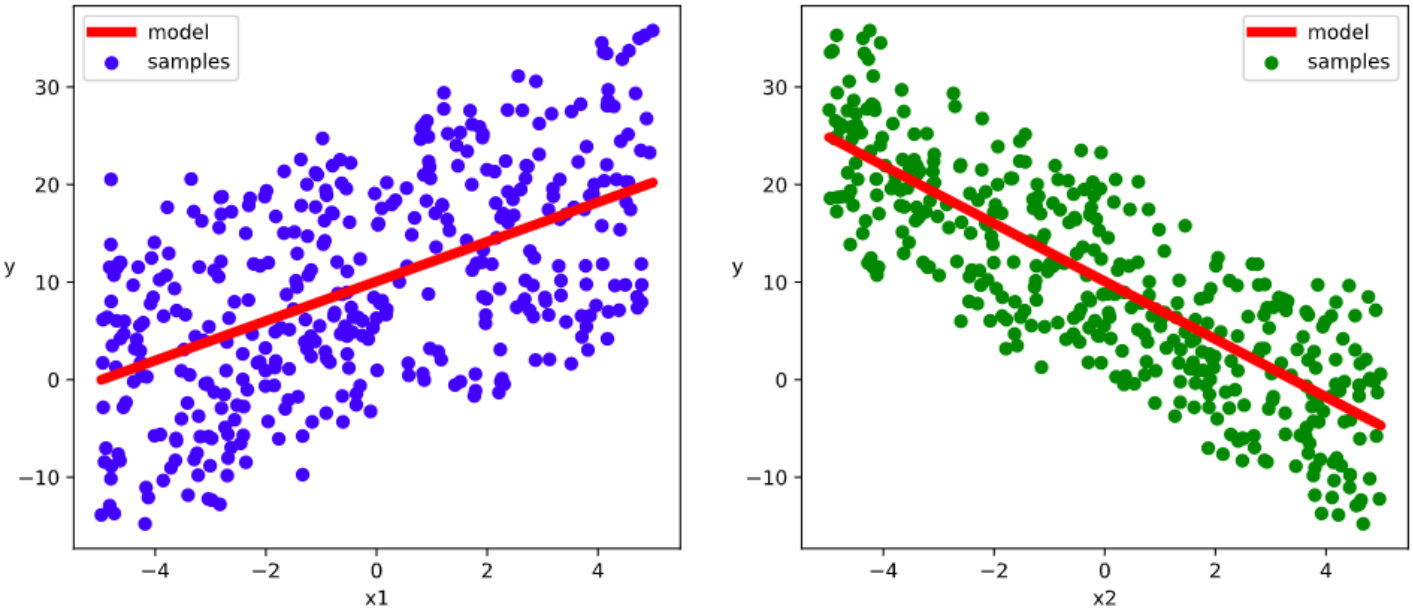
```
def train_step(model, features, labels):

    predictions = model.forward(features)
    loss = model.loss_func(predictions,labels)

    # 反向传播求梯度
    loss.backward()

    # 使用torch.no_grad()避免梯度记录，也可以通过操作 model.w.data 实现避免梯度记录
    with torch.no_grad():
        # 梯度下降法更新参数
        model.w -= 0.001*model.w.grad
        model.b -= 0.001*model.b.grad

        # 梯度清零
        model.w.grad.zero_()
        model.b.grad.zero_()
    return loss



# 测试train_step效果
batch_size = 10
(features,labels) = next(data_iter(X,Y,batch_size))
train_step(model,features,labels)
```

```
tensor(92.8199, grad_fn=<MeanBackward0>)
```

```
def train_model(model,epochs):
    for epoch in range(1,epochs+1):
        for features, labels in data_iter(X,Y,10):
            loss = train_step(model,features,labels)

        if epoch%200==0:
            printbar()
            print("epoch =",epoch,"loss = ",loss.item())
            print("model.w =",model.w.data)
            print("model.b =",model.b.data)

train_model(model,epochs = 1000)
```

```
================================================================================2020-07-05 08:27:57
epoch = 200 loss =  2.6340413093566895
model.w = tensor([[ 2.0283],
        [-2.9632]])
model.b = tensor([[10.0748]])

================================================================================2020-07-05 08:28:00
epoch = 400 loss =  2.24908709526062
model.w = tensor([[ 2.0300],
        [-2.9643]])
model.b = tensor([[10.0781]])

================================================================================2020-07-05 08:28:04
epoch = 600 loss =  1.510349154472351
model.w = tensor([[ 2.0290],
        [-2.9630]])
model.b = tensor([[10.0781]])

================================================================================2020-07-05 08:28:07
epoch = 800 loss =  1.038671851158142
model.w = tensor([[ 2.0314],
        [-2.9649]])
model.b = tensor([[10.0785]])

================================================================================2020-07-05 08:28:10
epoch = 1000 loss =  1.9742190837860107
model.w = tensor([[ 2.0313],
        [-2.9648]])
model.b = tensor([[10.0781]])
```

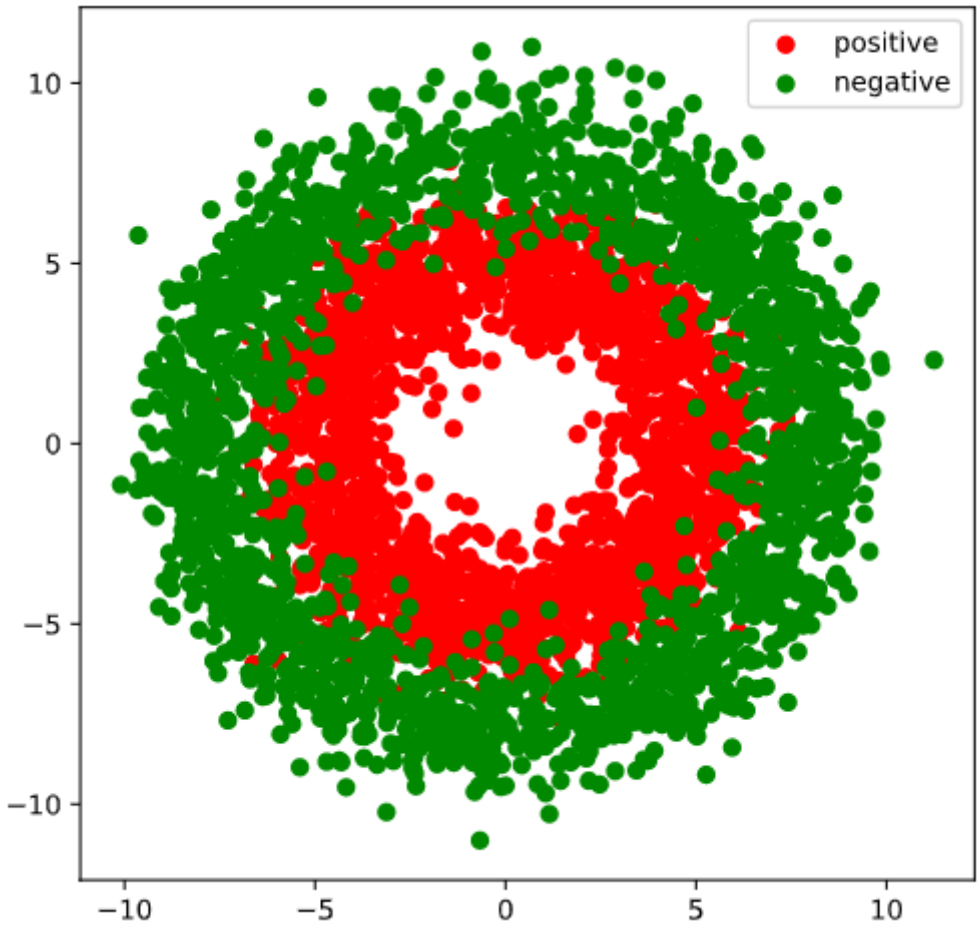
```
# 结果可视化

%matplotlib inline
%config InlineBackend.figure_format = 'svg'

plt.figure(figsize = (12,5))
ax1 = plt.subplot(121)
ax1.scatter(X[:,0].numpy(),Y[:,0].numpy(), c = "b",label = "samples")
ax1.plot(X[:,0].numpy(),(model.w[0].data*X[:,0]+model.b[0].data).numpy(),"-r",linewidth = 5.0,la
ax1.legend()
plt.xlabel("x1")
plt.ylabel("y",rotation = 0)


ax2 = plt.subplot(122)
ax2.scatter(X[:,1].numpy(),Y[:,0].numpy(), c = "g",label = "samples")
ax2.plot(X[:,1].numpy(),(model.w[1].data*X[:,1]+model.b[0].data).numpy(),"-r",linewidth = 5.0,la
ax2.legend()
plt.xlabel("x2")
plt.ylabel("y",rotation = 0)

plt.show()
```



二，DNN二分类模型

1，准备数据

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
import torch
from torch import nn
%matplotlib inline
%config InlineBackend.figure_format = 'svg'

#正负样本数量
n_positive,n_negative = 2000,2000

#生成正样本，小圆环分布
r_p = 5.0 + torch.normal(0.0,1.0,size = [n_positive,1])
theta_p = 2*np.pi*torch.rand([n_positive,1])
Xp = torch.cat([r_p*torch.cos(theta_p),r_p*torch.sin(theta_p)],axis = 1)
Yp = torch.ones_like(r_p)

#生成负样本，大圆环分布
r_n = 8.0 + torch.normal(0.0,1.0,size = [n_negative,1])
theta_n = 2*np.pi*torch.rand([n_negative,1])
Xn = torch.cat([r_n*torch.cos(theta_n),r_n*torch.sin(theta_n)],axis = 1)
Yn = torch.zeros_like(r_n)

#汇总样本
X = torch.cat([Xp,Xn],axis = 0)
Y = torch.cat([Yp,Yn],axis = 0)


#可视化
plt.figure(figsize = (6,6))
plt.scatter(Xp[:,0].numpy(),Xp[:,1].numpy(),c = "r")
plt.scatter(Xn[:,0].numpy(),Xn[:,1].numpy(),c = "g")
plt.legend(["positive","negative"]);
```




```
# 构建数据管道迭代器
def data_iter(features, labels, batch_size=8):
    num_examples = len(features)
    indices = list(range(num_examples))
    np.random.shuffle(indices)  #样本的读取顺序是随机的
    for i in range(0, num_examples, batch_size):
        indexs = torch.LongTensor(indices[i: min(i + batch_size, num_examples)])
        yield  features.index_select(0, indexs), labels.index_select(0, indexs)

# 测试数据管道效果
batch_size = 8
(features,labels) = next(data_iter(X,Y,batch_size))
print(features)
print(labels)
```

```
tensor([[ 6.9914, -1.0820],
        [ 4.8156,  4.0532],
        [-1.0697, -7.4644],
        [ 2.6291,  3.8851],
        [-1.6780, -4.3390],
        [-6.1495,  1.2269],
        [-4.3422,  3.9552],
        [-6.2265,  2.6159]])
tensor([[0.],
        [1.],
        [0.],
        [1.],
        [1.],
        [1.],
        [1.],
        [1.]])
```

2，定义模型

此处范例我们利用nn.Module来组织模型变量。

```
class DNNModel(nn.Module):
    def __init__(self):
        super(DNNModel, self).__init__()
        self.w1 = nn.Parameter(torch.randn(2,4))
        self.b1 = nn.Parameter(torch.zeros(1,4))
        self.w2 = nn.Parameter(torch.randn(4,8))
        self.b2 = nn.Parameter(torch.zeros(1,8))
        self.w3 = nn.Parameter(torch.randn(8,1))
        self.b3 = nn.Parameter(torch.zeros(1,1))

    # 正向传播
    def forward(self,x):
        x = torch.relu(x@self.w1 + self.b1)
        x = torch.relu(x@self.w2 + self.b2)
        y = torch.sigmoid(x@self.w3 + self.b3)
        return y

    # 损失函数(二元交叉熵)
    def loss_func(self,y_pred,y_true):
        #将预测值限制在1e-7以上, 1- (1e-7)以下，避免log(0)错误
        eps = 1e-7
        y_pred = torch.clamp(y_pred,eps,1.0-eps)
        bce = - y_true*torch.log(y_pred) - (1-y_true)*torch.log(1-y_pred)
        return torch.mean(bce)

    # 评估指标(准确率)
    def metric_func(self,y_pred,y_true):
        y_pred = torch.where(y_pred>0.5,torch.ones_like(y_pred,dtype = torch.float32),
                             torch.zeros_like(y_pred,dtype = torch.float32))
        acc = torch.mean(1-torch.abs(y_true-y_pred))
        return acc

model = DNNModel()



# 测试模型结构
batch_size = 10
(features,labels) = next(data_iter(X,Y,batch_size))

predictions = model(features)

loss = model.loss_func(labels,predictions)
metric = model.metric_func(labels,predictions)

print("init loss:", loss.item())
print("init metric:", metric.item())
```

```
init loss: 7.979694366455078
init metric: 0.50347900390625
```

```
len(list(model.parameters()))
```

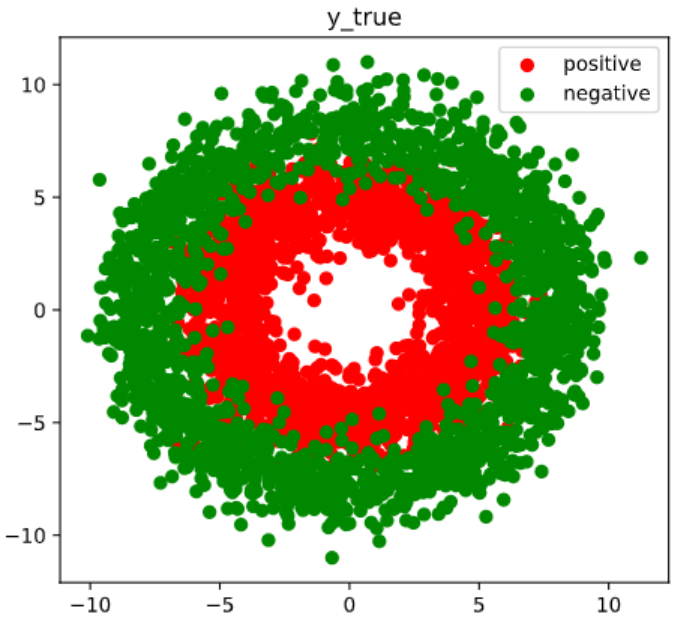
```
6
```

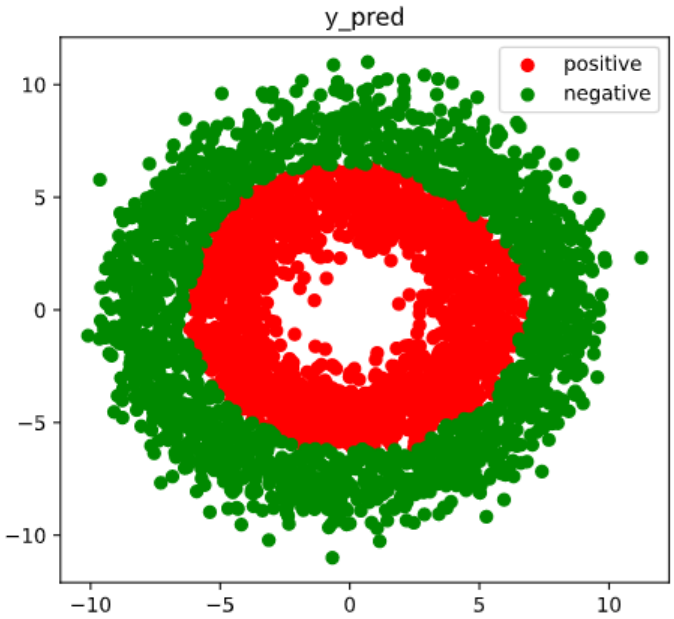
3，训练模型

```
def train_step(model, features, labels):

    # 正向传播求损失
    predictions = model.forward(features)
    loss = model.loss_func(predictions,labels)
    metric = model.metric_func(predictions,labels)

    # 反向传播求梯度
    loss.backward()

    # 梯度下降法更新参数
    for param in model.parameters():
        #注意是对param.data进行重新赋值,避免此处操作引起梯度记录
        param.data = (param.data - 0.01*param.grad.data)

    # 梯度清零
    model.zero_grad()

    return loss.item(),metric.item()


def train_model(model,epochs):
    for epoch in range(1,epochs+1):
        loss_list,metric_list = [],[]
        for features, labels in data_iter(X,Y,20):
            lossi,metrici = train_step(model,features,labels)
            loss_list.append(lossi)
            metric_list.append(metrici)
        loss = np.mean(loss_list)
        metric = np.mean(metric_list)

        if epoch%100==0:
            printbar()
            print("epoch =",epoch,"loss = ",loss,"metric = ",metric)

train_model(model,epochs = 1000)
```

```
==========================================================================================2020-07-05 08:32:16
epoch = 100 loss =  0.24841043589636683 metric =  0.8944999960064888

==========================================================================================2020-07-05 08:32:34
epoch = 200 loss =  0.20398724960163236 metric =  0.920999992787838

==========================================================================================2020-07-05 08:32:54
epoch = 300 loss =  0.19509393003769218 metric =  0.9239999914169311

==========================================================================================2020-07-05 08:33:14
epoch = 400 loss =  0.19067603485658766 metric =  0.9272499939799309

==========================================================================================2020-07-05 08:33:33
epoch = 500 loss =  0.1898010154720396 metric =  0.9237499925494194

==========================================================================================2020-07-05 08:33:54
epoch = 600 loss =  0.19151576517149807 metric =  0.9254999926686287

==========================================================================================2020-07-05 08:34:18
epoch = 700 loss =  0.18914461021777243 metric =  0.9274999949336052

==========================================================================================2020-07-05 08:34:39
epoch = 800 loss =  0.18801998342387377 metric =  0.9264999932050705

==========================================================================================2020-07-05 08:35:00
epoch = 900 loss =  0.1852504052128643 metric =  0.9249999937415123

==========================================================================================2020-07-05 08:35:21
epoch = 1000 loss =  0.18695520935580134 metric =  0.9272499927878379
```

```
# 结果可视化
fig, (ax1,ax2) = plt.subplots(nrows=1,ncols=2,figsize = (12,5))
ax1.scatter(Xp[:,0],Xp[:,1], c="r")
ax1.scatter(Xn[:,0],Xn[:,1],c = "g")
ax1.legend(["positive","negative"]);
ax1.set_title("y_true");

Xp_pred = X[torch.squeeze(model.forward(X)>=0.5)]
Xn_pred = X[torch.squeeze(model.forward(X)<0.5)]

ax2.scatter(Xp_pred[:,0],Xp_pred[:,1],c = "r")
ax2.scatter(Xn_pred[:,0],Xn_pred[:,1],c = "g")
ax2.legend(["positive","negative"]);
ax2.set_title("y_pred");
```





3-2,中阶API示范

下面的范例使用Pytorch的中阶API实现线性回归模型和和DNN二分类模型。

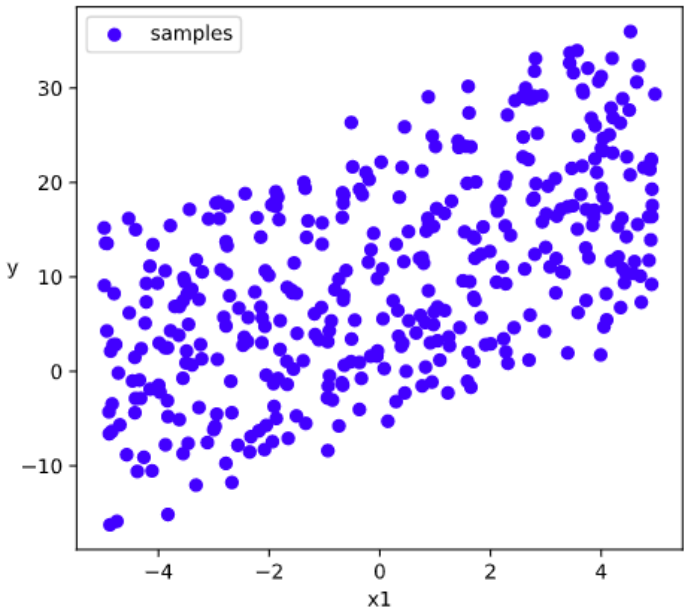
Pytorch的中阶API主要包括各种模型层，损失函数，优化器，数据管道等等。

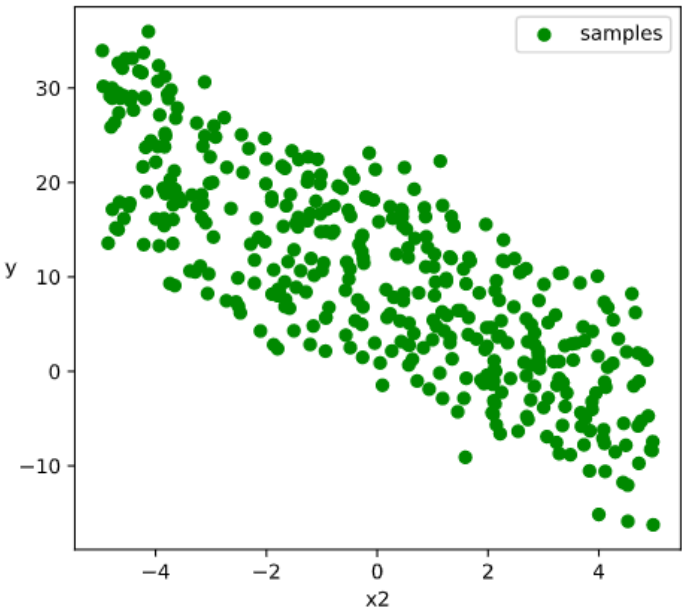
```
import os
import datetime

#打印时间
def printbar():
    nowtime = datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S')
    print("\n"+"=========="*8 + "%s"%nowtime)

#mac系统上pytorch和matplotlib在jupyter中同时跑需要更改环境变量
os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"
```

一，线性回归模型

1，准备数据

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
import torch
from torch import nn
import torch.nn.functional as F
from torch.utils.data import Dataset,DataLoader,TensorDataset

#样本数量
n = 400

# 生成测试用数据集
X = 10*torch.rand([n,2])-5.0  #torch.rand是均匀分布
w0 = torch.tensor([[2.0],[-3.0]])
b0 = torch.tensor([[10.0]])
Y = X@w0 + b0 + torch.normal( 0.0,2.0,size = [n,1])  # @表示矩阵乘法,增加正态扰动



# 数据可视化

%matplotlib inline
%config InlineBackend.figure_format = 'svg'

plt.figure(figsize = (12,5))
ax1 = plt.subplot(121)
ax1.scatter(X[:,0],Y[:,0], c = "b",label = "samples")
ax1.legend()
plt.xlabel("x1")
plt.ylabel("y",rotation = 0)

ax2 = plt.subplot(122)
ax2.scatter(X[:,1],Y[:,0], c = "g",label = "samples")
ax2.legend()
plt.xlabel("x2")
plt.ylabel("y",rotation = 0)
plt.show()
```






```
#构建输入数据管道
ds = TensorDataset(X,Y)
dl = DataLoader(ds,batch_size = 10,shuffle=True,num_workers=2)
```

2，定义模型

```
model = nn.Linear(2,1) #线性层

model.loss_func = nn.MSELoss()
model.optimizer = torch.optim.SGD(model.parameters(),lr = 0.01)
```

3，训练模型

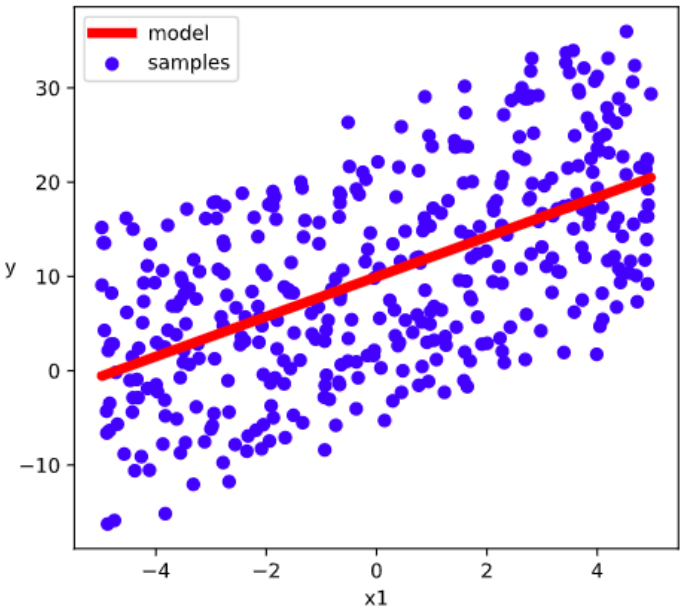
```
def train_step(model, features, labels):

    predictions = model(features)
    loss = model.loss_func(predictions,labels)
    loss.backward()
    model.optimizer.step()
    model.optimizer.zero_grad()
    return loss.item()

# 测试train_step效果
features,labels = next(iter(dl))
train_step(model,features,labels)
```

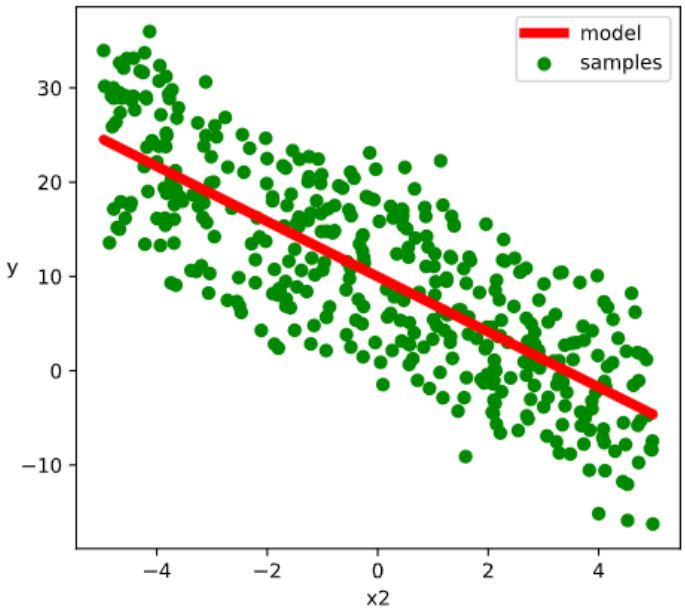
```
269.98016357421875
```

```
def train_model(model,epochs):
    for epoch in range(1,epochs+1):
        for features, labels in dl:
            loss = train_step(model,features,labels)
        if epoch%50==0:
            printbar()
            w = model.state_dict()["weight"]
            b = model.state_dict()["bias"]
            print("epoch =",epoch,"loss = ",loss)
            print("w =",w)
            print("b =",b)
train_model(model,epochs = 200)
```

```
==========2020-07-05 22:51:53
epoch = 50 loss =  3.0177409648895264
w = tensor([[ 1.9315, -2.9573]])
b = tensor([9.9625])

==========2020-07-05 22:51:57
epoch = 100 loss =  2.1144354343414307
w = tensor([[ 1.9760, -2.9398]])
b = tensor([9.9428])

==========2020-07-05 22:52:01
epoch = 150 loss =  3.290461778640747
w = tensor([[ 2.1075, -2.9509]])
b = tensor([9.9599])

==========2020-07-05 22:52:06
epoch = 200 loss =  3.047853469848633
w = tensor([[ 2.1134, -2.9306]])
b = tensor([9.9722])
```

```
# 结果可视化

%matplotlib inline
%config InlineBackend.figure_format = 'svg'

w,b = model.state_dict()["weight"],model.state_dict()["bias"]

plt.figure(figsize = (12,5))
ax1 = plt.subplot(121)
ax1.scatter(X[:,0],Y[:,0], c = "b",label = "samples")
ax1.plot(X[:,0],w[0,0]*X[:,0]+b[0],"-r",linewidth = 5.0,label = "model")
ax1.legend()
plt.xlabel("x1")
plt.ylabel("y",rotation = 0)



ax2 = plt.subplot(122)
ax2.scatter(X[:,1],Y[:,0], c = "g",label = "samples")
ax2.plot(X[:,1],w[0,1]*X[:,1]+b[0],"-r",linewidth = 5.0,label = "model")
ax2.legend()
plt.xlabel("x2")
plt.ylabel("y",rotation = 0)

plt.show()
```

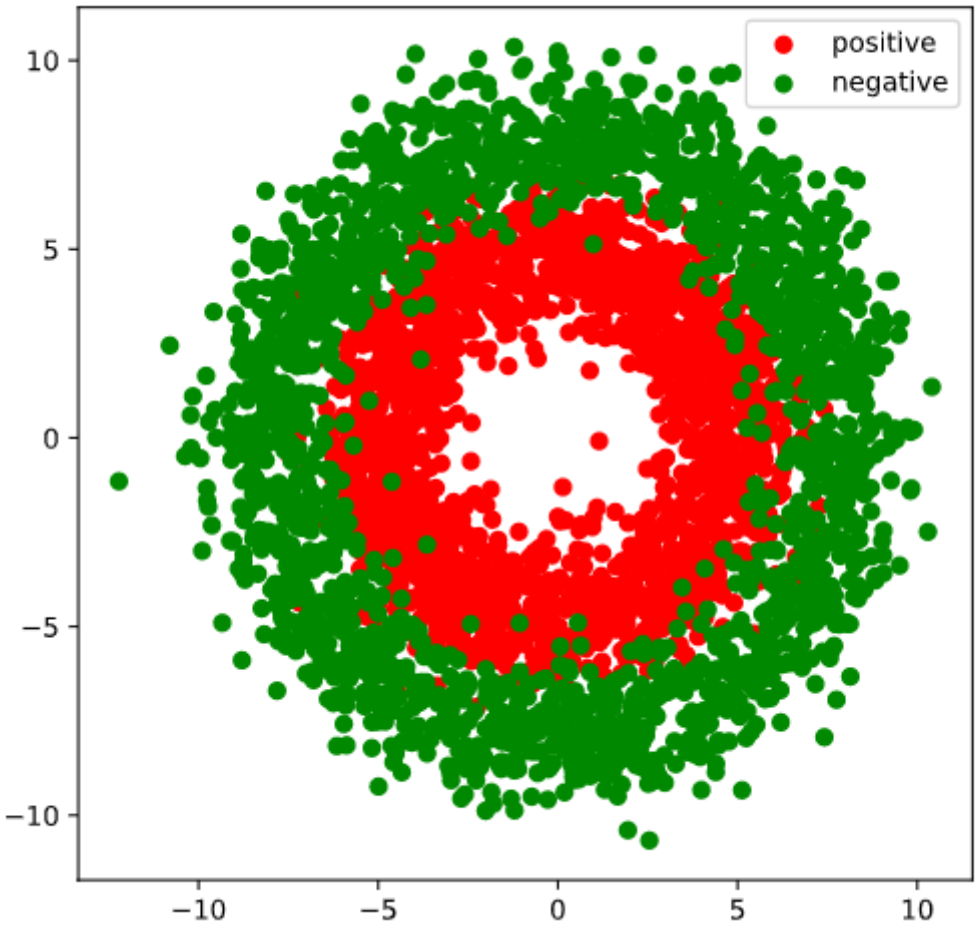




二， DNN二分类模型

1，准备数据

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
import torch
from torch import nn
import torch.nn.functional as F
from torch.utils.data import Dataset,DataLoader,TensorDataset
%matplotlib inline
%config InlineBackend.figure_format = 'svg'

#正负样本数量
n_positive,n_negative = 2000,2000

#生成正样本，小圆环分布
r_p = 5.0 + torch.normal(0.0,1.0,size = [n_positive,1])
theta_p = 2*np.pi*torch.rand([n_positive,1])
Xp = torch.cat([r_p*torch.cos(theta_p),r_p*torch.sin(theta_p)],axis = 1)
Yp = torch.ones_like(r_p)

#生成负样本，大圆环分布
r_n = 8.0 + torch.normal(0.0,1.0,size = [n_negative,1])
theta_n = 2*np.pi*torch.rand([n_negative,1])
Xn = torch.cat([r_n*torch.cos(theta_n),r_n*torch.sin(theta_n)],axis = 1)
Yn = torch.zeros_like(r_n)

#汇总样本
X = torch.cat([Xp,Xn],axis = 0)
Y = torch.cat([Yp,Yn],axis = 0)


#可视化
plt.figure(figsize = (6,6))
plt.scatter(Xp[:,0],Xp[:,1],c = "r")
plt.scatter(Xn[:,0],Xn[:,1],c = "g")
plt.legend(["positive","negative"]);
```




```
#构建输入数据管道
ds = TensorDataset(X,Y)
dl = DataLoader(ds,batch_size = 10,shuffle=True,num_workers=2)
```

2, 定义模型

```
class DNNModel(nn.Module):
    def __init__(self):
        super(DNNModel, self).__init__()
        self.fc1 = nn.Linear(2,4)
        self.fc2 = nn.Linear(4,8)
        self.fc3 = nn.Linear(8,1)

    # 正向传播
    def forward(self,x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        y = nn.Sigmoid()(self.fc3(x))
        return y

    # 损失函数
    def loss_func(self,y_pred,y_true):
        return nn.BCELoss()(y_pred,y_true)

    # 评估函数(准确率)
    def metric_func(self,y_pred,y_true):
        y_pred = torch.where(y_pred>0.5,torch.ones_like(y_pred,dtype = torch.float32),
                             torch.zeros_like(y_pred,dtype = torch.float32))
        acc = torch.mean(1-torch.abs(y_true-y_pred))
        return acc

    # 优化器
    @property
    def optimizer(self):
        return torch.optim.Adam(self.parameters(),lr = 0.001)

model = DNNModel()



# 测试模型结构
(features,labels) = next(iter(dl))
predictions = model(features)

loss = model.loss_func(predictions,labels)
metric = model.metric_func(predictions,labels)

print("init loss:",loss.item())
print("init metric:",metric.item())
```

```
init loss: 0.7065666913986206
init metric: 0.6000000238418579
```

3，训练模型

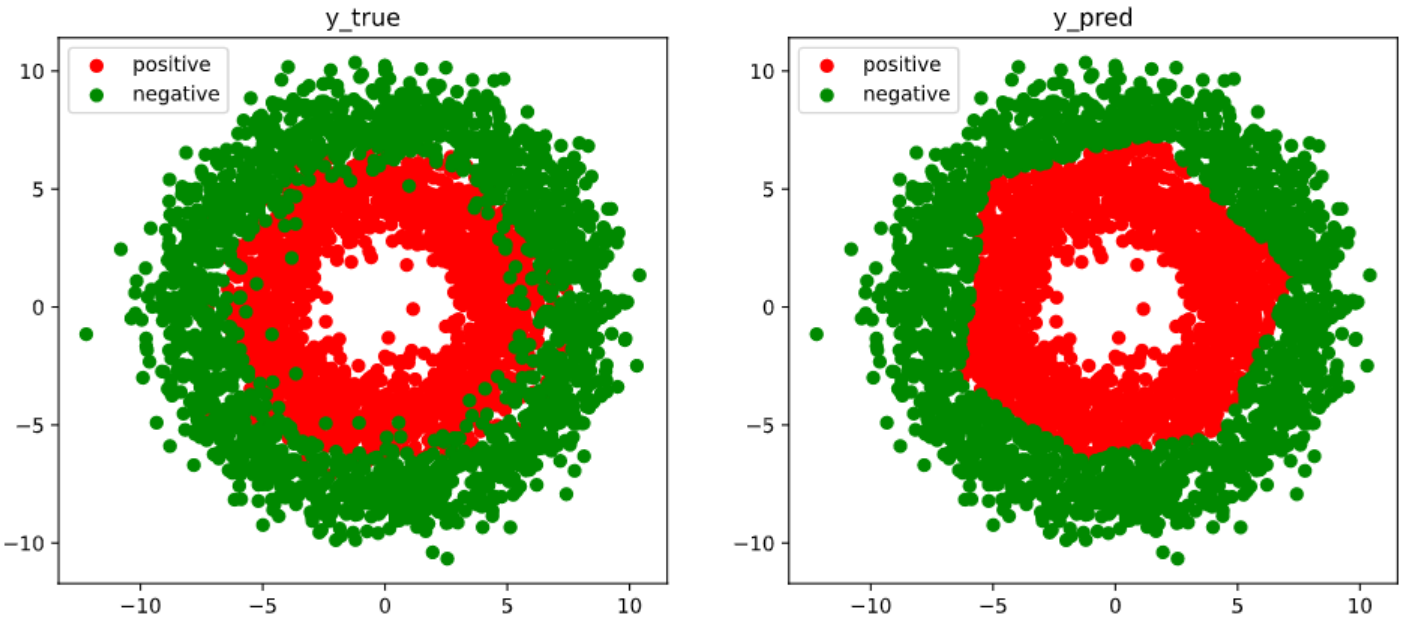
```
def train_step(model, features, labels):

    # 正向传播求损失
    predictions = model(features)
    loss = model.loss_func(predictions,labels)
    metric = model.metric_func(predictions,labels)

    # 反向传播求梯度
    loss.backward()

    # 更新模型参数
    model.optimizer.step()
    model.optimizer.zero_grad()

    return loss.item(),metric.item()

# 测试train_step效果
features,labels = next(iter(dl))
train_step(model,features,labels)
```

```
(0.6048880815505981, 0.699999988079071)
```

```
def train_model(model,epochs):
    for epoch in range(1,epochs+1):
        loss_list,metric_list = [],[]
        for features, labels in dl:
            lossi,metrici = train_step(model,features,labels)
            loss_list.append(lossi)
            metric_list.append(metrici)
        loss = np.mean(loss_list)
        metric = np.mean(metric_list)

        if epoch%100==0:
            printbar()
            print("epoch =",epoch,"loss = ",loss,"metric = ",metric)

train_model(model,epochs = 300)
```

```
==========================================================================================2020-07-05 22:56:38
epoch = 100 loss =  0.23532892110607917 metric =  0.934749992787838

==========================================================================================2020-07-05 22:58:18
epoch = 200 loss =  0.24743918558603128 metric =  0.934999993443489

==========================================================================================2020-07-05 22:59:56
epoch = 300 loss =  0.2936080049697884 metric =  0.931499992609024
```

```
# 结果可视化
fig, (ax1,ax2) = plt.subplots(nrows=1,ncols=2,figsize = (12,5))
ax1.scatter(Xp[:,0],Xp[:,1], c="r")
ax1.scatter(Xn[:,0],Xn[:,1],c = "g")
ax1.legend(["positive","negative"]);
ax1.set_title("y_true");

Xp_pred = X[torch.squeeze(model.forward(X)>=0.5)]
Xn_pred = X[torch.squeeze(model.forward(X)<0.5)]

ax2.scatter(Xp_pred[:,0],Xp_pred[:,1],c = "r")
ax2.scatter(Xn_pred[:,0],Xn_pred[:,1],c = "g")
ax2.legend(["positive","negative"]);
ax2.set_title("y_pred");
```



3-3,高阶API示范

Pytorch没有官方的高阶API，一般需要用户自己实现训练循环、验证循环、和预测循环。

作者通过仿照tf.keras.Model的功能对Pytorch的nn.Module进行了封装，

实现了 fit, validate，predict, summary 方法，相当于用户自定义高阶API。

并在其基础上实现线性回归模型和DNN二分类模型。

```
import os
import datetime
from torchkeras import Model, summary

#打印时间
def printbar():
    nowtime = datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S')
    print("\n"+"=========="*8 + "%s"%nowtime)

#mac系统上pytorch和matplotlib在jupyter中同时跑需要更改环境变量
os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"
```

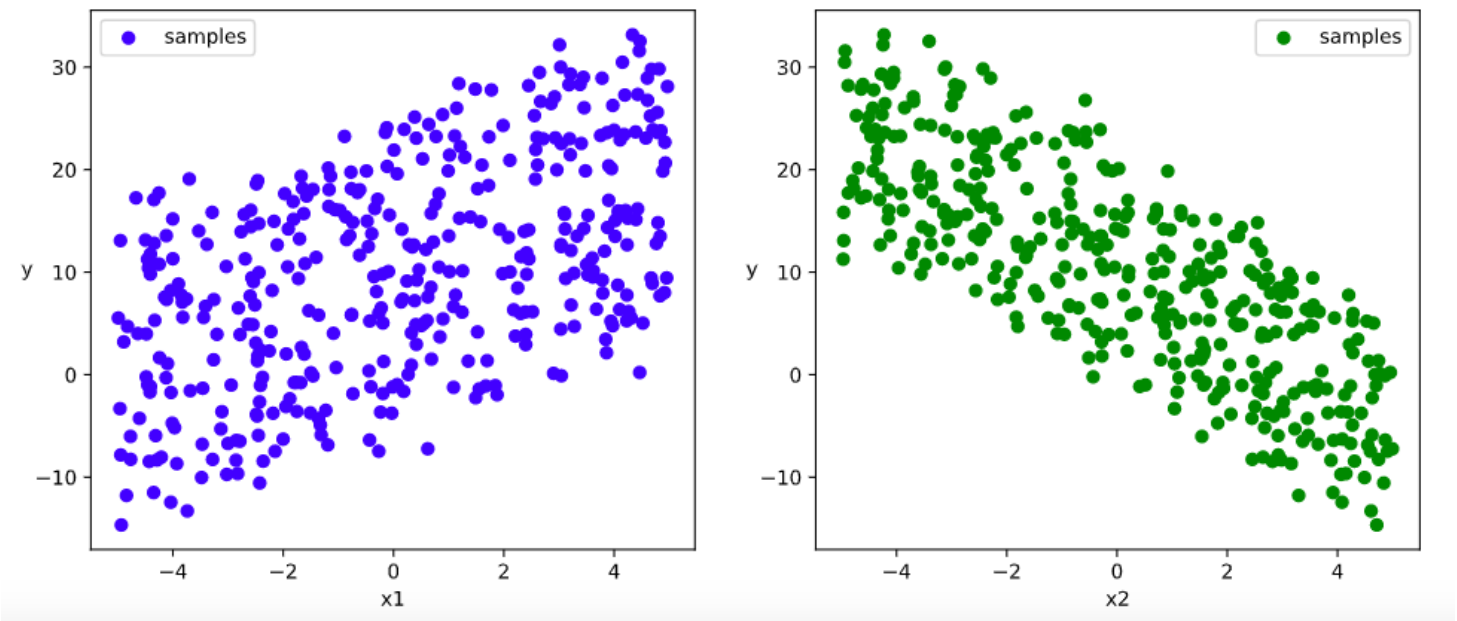
一，线性回归模型

此范例我们通过继承上述用户自定义 Model模型接口，实现线性回归模型。

1，准备数据

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
import torch
from torch import nn
import torch.nn.functional as F
from torch.utils.data import Dataset,DataLoader,TensorDataset

#样本数量
n = 400

# 生成测试用数据集
X = 10*torch.rand([n,2])-5.0  #torch.rand是均匀分布
w0 = torch.tensor([[2.0],[-3.0]])
b0 = torch.tensor([[10.0]])
Y = X@w0 + b0 + torch.normal( 0.0,2.0,size = [n,1])  # @表示矩阵乘法,增加正态扰动
```

```
# 数据可视化

%matplotlib inline
%config InlineBackend.figure_format = 'svg'

plt.figure(figsize = (12,5))
ax1 = plt.subplot(121)
ax1.scatter(X[:,0],Y[:,0], c = "b",label = "samples")
ax1.legend()
plt.xlabel("x1")
plt.ylabel("y",rotation = 0)

ax2 = plt.subplot(122)
ax2.scatter(X[:,1],Y[:,0], c = "g",label = "samples")
ax2.legend()
plt.xlabel("x2")
plt.ylabel("y",rotation = 0)
plt.show()
```




```
#构建输入数据管道
ds = TensorDataset(X,Y)
ds_train,ds_valid = torch.utils.data.random_split(ds,[int(400*0.7),400-int(400*0.7)])
dl_train = DataLoader(ds_train,batch_size = 10,shuffle=True,num_workers=2)
dl_valid = DataLoader(ds_valid,batch_size = 10,num_workers=2)
```

2，定义模型

```
# 继承用户自定义模型
from torchkeras import Model
class LinearRegression(Model):
    def __init__(self):
        super(LinearRegression, self).__init__()
        self.fc = nn.Linear(2,1)

    def forward(self,x):
        return self.fc(x)

model = LinearRegression()



model.summary(input_shape = (2,))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1                    [-1, 1]               3
================================================================
Total params: 3
Trainable params: 3
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.000008
Forward/backward pass size (MB): 0.000008
Params size (MB): 0.000011
Estimated Total Size (MB): 0.000027
----------------------------------------------------------------
```

3，训练模型

使用fit方法进行训练

```
def mean_absolute_error(y_pred,y_true):
    return torch.mean(torch.abs(y_pred-y_true))

def mean_absolute_percent_error(y_pred,y_true):
    absolute_percent_error = (torch.abs(y_pred-y_true)+1e-7)/(torch.abs(y_true)+1e-7)
    return torch.mean(absolute_percent_error)

model.compile(loss_func = nn.MSELoss(),
              optimizer= torch.optim.Adam(model.parameters(),lr = 0.01),
              metrics_dict={"mae":mean_absolute_error,"mape":mean_absolute_percent_error})

dfhistory = model.fit(200,dl_train = dl_train, dl_val = dl_valid,log_step_freq = 20)
```

```
Start Training ...

==========================================================================================2020-07-05 23:07:25
{'step': 20, 'loss': 226.768, 'mae': 12.198, 'mape': 1.212}

 +-------+---------+-------+-------+----------+---------+----------+
| epoch |   loss  |  mae  |  mape | val_loss | val_mae | val_mape |
+-------+---------+-------+-------+----------+---------+----------+
|   1   | 230.773 | 12.41 | 1.394 | 223.262  |  12.582 |  1.095   |
+-------+---------+-------+-------+----------+---------+----------+

==========================================================================================2020-07-05 23:07:26
{'step': 20, 'loss': 200.964, 'mae': 11.584, 'mape': 1.382}

 +-------+---------+--------+------+----------+---------+----------+
| epoch |   loss  |  mae   | mape | val_loss | val_mae | val_mape |
+-------+---------+--------+------+----------+---------+----------+
|   2   | 206.238 | 11.759 | 1.26 | 199.669  |  11.895 |  1.012   |
+-------+---------+--------+------+----------+---------+----------+

==========================================================================================2020-07-05 23:07:26
{'step': 20, 'loss': 188.247, 'mae': 11.387, 'mape': 1.172}

 +-------+---------+--------+-------+----------+---------+----------+
| epoch |   loss  |  mae   |  mape | val_loss | val_mae | val_mape |
+-------+---------+--------+-------+----------+---------+----------+
|   3   | 185.185 | 11.177 | 1.189 | 178.112  |  11.24  |  0.952   |
+-------+---------+--------+-------+----------+---------+----------+
==========================================================================================2020-07-05 23:07:59
{'step': 20, 'loss': 4.14, 'mae': 1.677, 'mape': 1.845}

 +-------+-------+-------+-------+----------+---------+----------+
| epoch |  loss |  mae  |  mape | val_loss | val_mae | val_mape |
+-------+-------+-------+-------+----------+---------+----------+
|  199  | 4.335 | 1.707 | 1.441 |  3.741   |  1.533  |  0.359   |
+-------+-------+-------+-------+----------+---------+----------+

==========================================================================================2020-07-05 23:07:59
{'step': 20, 'loss': 4.653, 'mae': 1.775, 'mape': 0.679}

 +-------+------+-------+-------+----------+---------+----------+
| epoch | loss |  mae  |  mape | val_loss | val_mae | val_mape |
+-------+------+-------+-------+----------+---------+----------+
|  200  | 4.37 | 1.718 | 1.394 |  3.749   |  1.534  |  0.363   |
+-------+------+-------+-------+----------+---------+----------+

==========================================================================================2020-07-05 23:07:59
Finished Training...
```

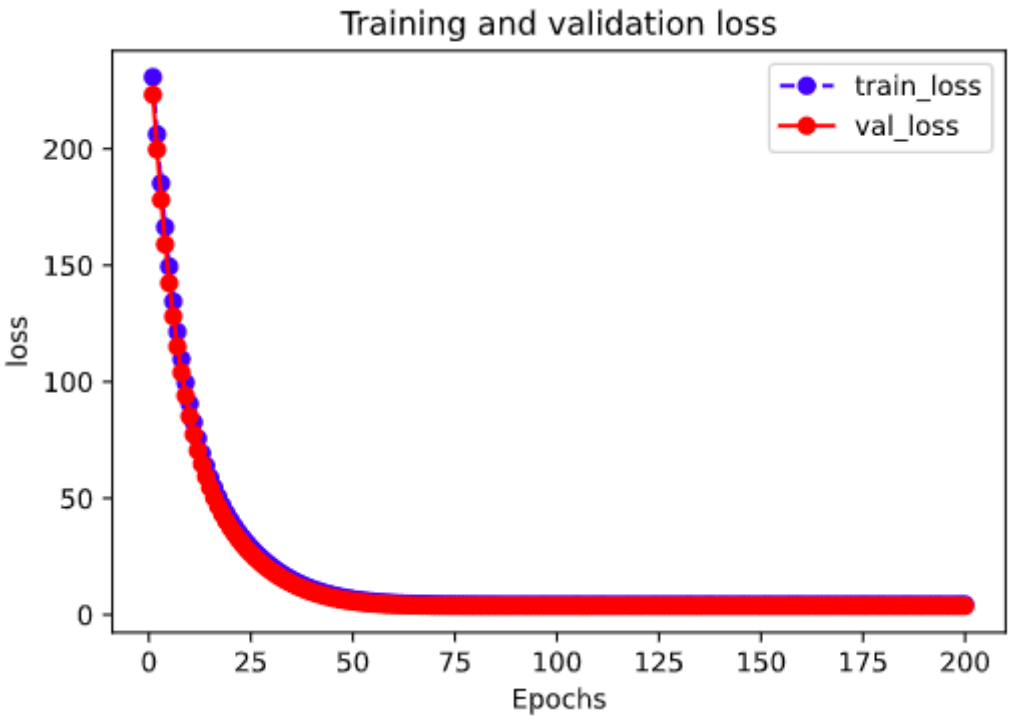
```
# 结果可视化

%matplotlib inline
%config InlineBackend.figure_format = 'svg'

w,b = model.state_dict()["fc.weight"],model.state_dict()["fc.bias"]

plt.figure(figsize = (12,5))
ax1 = plt.subplot(121)
ax1.scatter(X[:,0],Y[:,0], c = "b",label = "samples")
ax1.plot(X[:,0],w[0,0]*X[:,0]+b[0],"-r",linewidth = 5.0,label = "model")
ax1.legend()
plt.xlabel("x1")
plt.ylabel("y",rotation = 0)


ax2 = plt.subplot(122)
ax2.scatter(X[:,1],Y[:,0], c = "g",label = "samples")
ax2.plot(X[:,1],w[0,1]*X[:,1]+b[0],"-r",linewidth = 5.0,label = "model")
ax2.legend()
plt.xlabel("x2")
plt.ylabel("y",rotation = 0)

plt.show()
```

4，评估模型

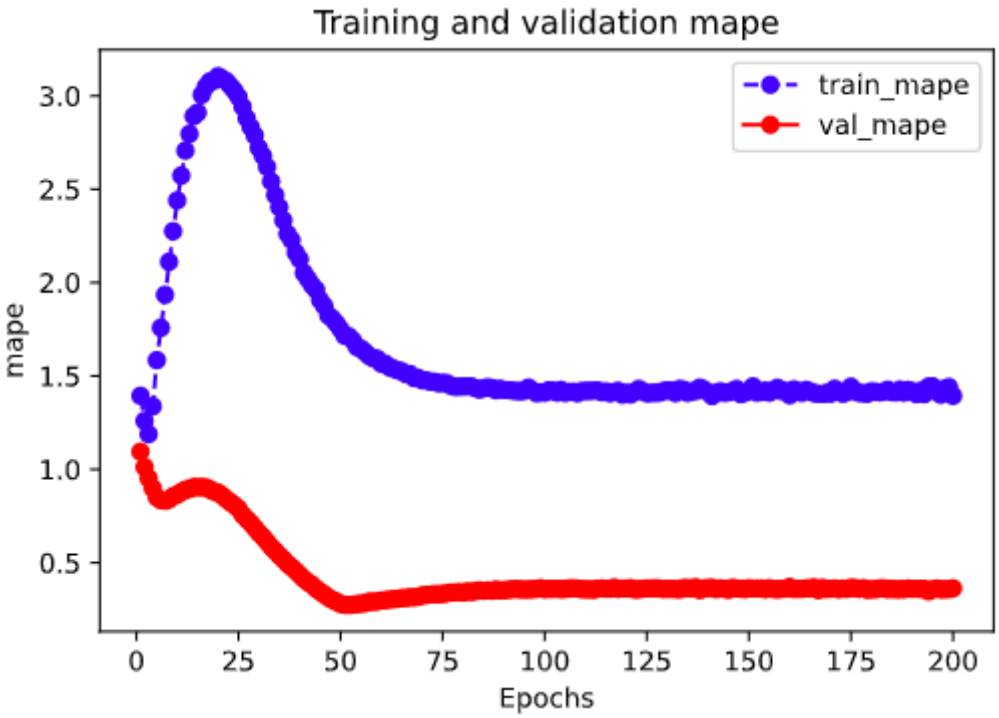
```
dfhistory.tail()
```

	loss	mae	mape	val_loss	val_mae	val_mape
195	4.317858	1.703719	1.417378	3.755003	1.533518	0.362025
196	4.332180	1.710020	1.399523	3.742664	1.530505	0.357305
197	4.336281	1.708945	1.431962	3.731560	1.529354	0.357993
198	4.335447	1.706877	1.440796	3.740713	1.532849	0.358673
199	4.370343	1.717717	1.393625	3.749117	1.533614	0.363198

```
%matplotlib inline
%config InlineBackend.figure_format = 'svg'

import matplotlib.pyplot as plt

def plot_metric(dfhistory, metric):
    train_metrics = dfhistory[metric]
    val_metrics = dfhistory['val_'+metric]
    epochs = range(1, len(train_metrics) + 1)
    plt.plot(epochs, train_metrics, 'bo--')
    plt.plot(epochs, val_metrics, 'ro-')
    plt.title('Training and validation '+ metric)
    plt.xlabel("Epochs")
    plt.ylabel(metric)
    plt.legend(["train_"+metric, 'val_'+metric])
    plt.show()



plot_metric(dfhistory,"loss")
```




```
plot_metric(dfhistory,"mape")
```




```
# 评估
model.evaluate(dl_valid)
```

```
{'val_loss': 3.749117374420166,
 'val_mae': 1.5336137612660725,
 'val_mape': 0.36319838215907413}
```

5，使用模型

```
# 预测
dl = DataLoader(TensorDataset(X))
model.predict(dl)[0:10]
```

```
tensor([[ 3.9212],
        [ 8.6336],
        [ 6.1982],
        [ 6.1212],
        [-5.0974],
        [-6.3183],
        [ 4.6588],
        [ 5.5349],
        [11.9106],
        [24.6937]])
```

```
# 预测
model.predict(dl_valid)[0:10]
```

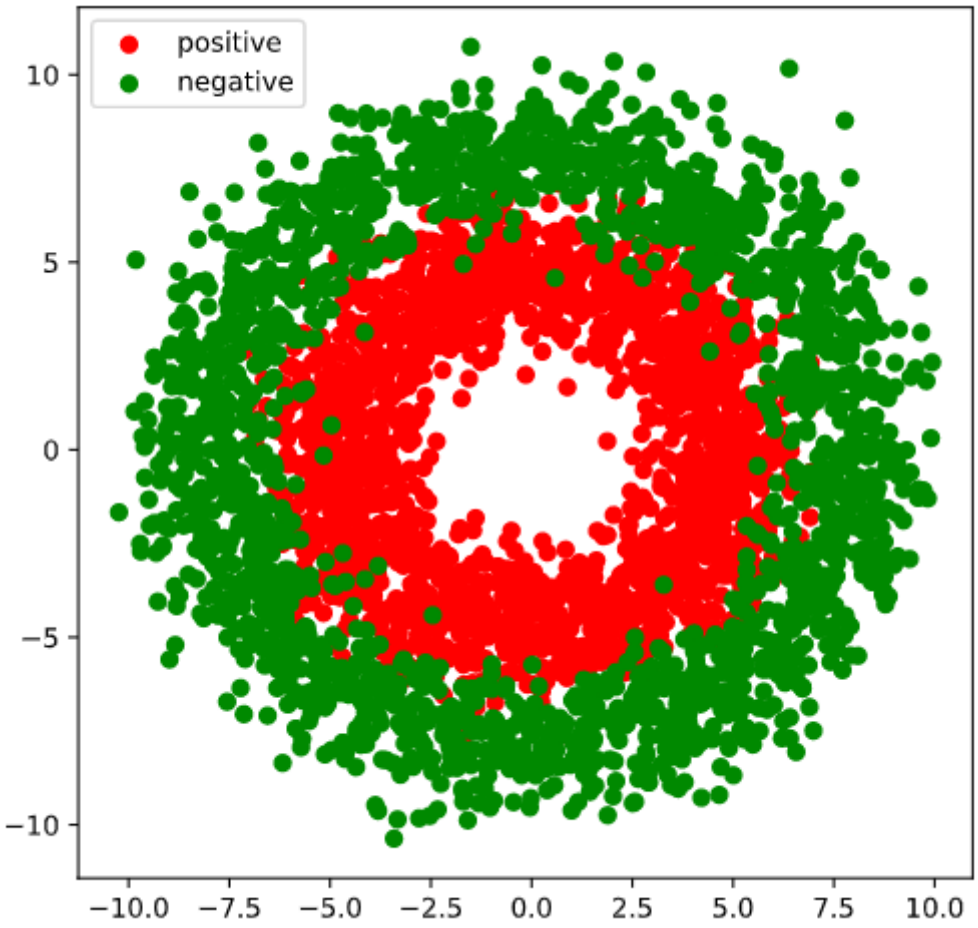
```
tensor([[ 2.8368],
        [16.2797],
        [ 2.3135],
        [ 9.5395],
        [16.4363],
        [10.0742],
        [15.0864],
        [12.9775],
        [21.8568],
        [21.8226]])
```

二，DNN二分类模型

此范例我们通过继承上述用户自定义 Model模型接口，实现DNN二分类模型。

1，准备数据

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
import torch
from torch import nn
import torch.nn.functional as F
from torch.utils.data import Dataset,DataLoader,TensorDataset
import torchkeras
%matplotlib inline
%config InlineBackend.figure_format = 'svg'

#正负样本数量
n_positive,n_negative = 2000,2000

#生成正样本，小圆环分布
r_p = 5.0 + torch.normal(0.0,1.0,size = [n_positive,1])
theta_p = 2*np.pi*torch.rand([n_positive,1])
Xp = torch.cat([r_p*torch.cos(theta_p),r_p*torch.sin(theta_p)],axis = 1)
Yp = torch.ones_like(r_p)

#生成负样本，大圆环分布
r_n = 8.0 + torch.normal(0.0,1.0,size = [n_negative,1])
theta_n = 2*np.pi*torch.rand([n_negative,1])
Xn = torch.cat([r_n*torch.cos(theta_n),r_n*torch.sin(theta_n)],axis = 1)
Yn = torch.zeros_like(r_n)

#汇总样本
X = torch.cat([Xp,Xn],axis = 0)
Y = torch.cat([Yp,Yn],axis = 0)


#可视化
plt.figure(figsize = (6,6))
plt.scatter(Xp[:,0],Xp[:,1],c = "r")
plt.scatter(Xn[:,0],Xn[:,1],c = "g")
plt.legend(["positive","negative"]);
```




```
ds = TensorDataset(X,Y)

ds_train,ds_valid = torch.utils.data.random_split(ds,[int(len(ds)*0.7),len(ds)-int(len(ds)*0.7)])
dl_train = DataLoader(ds_train,batch_size = 100,shuffle=True,num_workers=2)
dl_valid = DataLoader(ds_valid,batch_size = 100,num_workers=2)
```

2，定义模型

```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(2,4)
        self.fc2 = nn.Linear(4,8)
        self.fc3 = nn.Linear(8,1)

    def forward(self,x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        y = nn.Sigmoid()(self.fc3(x))
        return y

model = torchkeras.Model(Net())
model.summary(input_shape =(2,))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1                    [-1, 4]              12
            Linear-2                    [-1, 8]              40
            Linear-3                    [-1, 1]               9
================================================================
Total params: 61
Trainable params: 61
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.000008
Forward/backward pass size (MB): 0.000099
Params size (MB): 0.000233
Estimated Total Size (MB): 0.000340
----------------------------------------------------------------
```

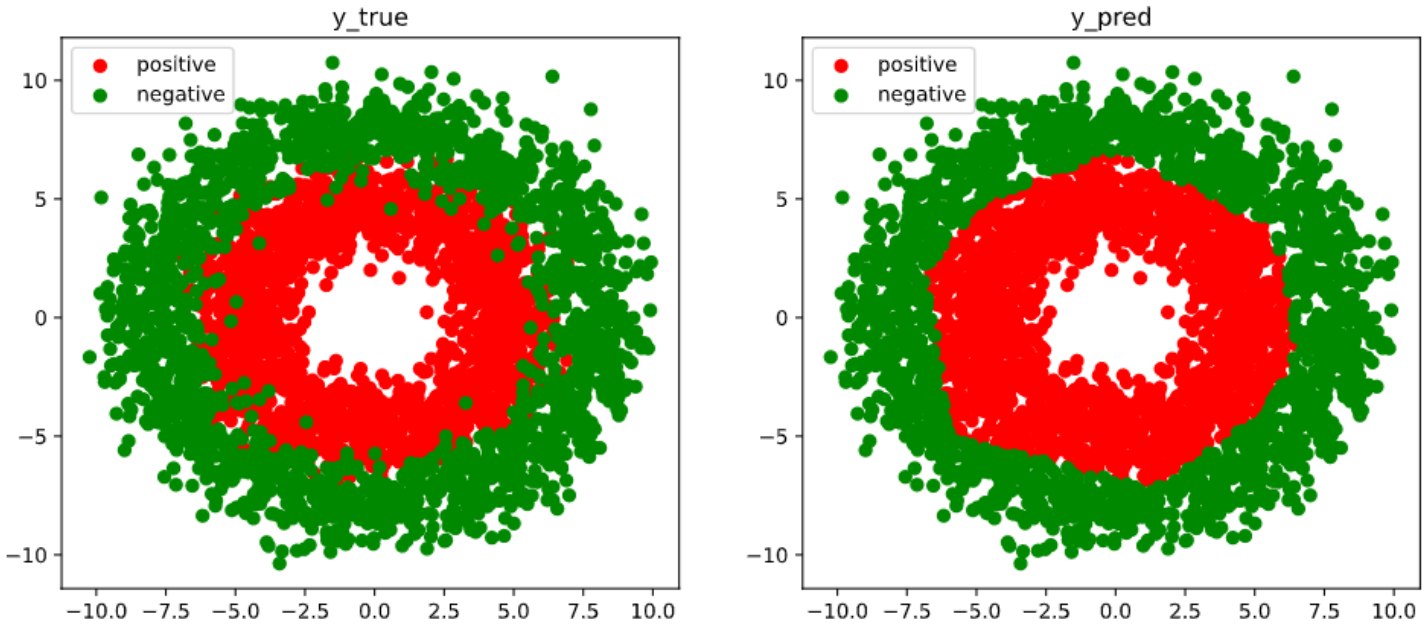
3，训练模型

```
# 准确率
def accuracy(y_pred,y_true):
    y_pred = torch.where(y_pred>0.5,torch.ones_like(y_pred,dtype = torch.float32),
                         torch.zeros_like(y_pred,dtype = torch.float32))
    acc = torch.mean(1-torch.abs(y_true-y_pred))
    return acc

model.compile(loss_func = nn.BCELoss(),optimizer= torch.optim.Adam(model.parameters(),lr = 0.01)
              metrics_dict={"accuracy":accuracy})

dfhistory = model.fit(100,dl_train = dl_train,dl_val = dl_valid,log_step_freq = 10)
```

```
Start Training ...

==========================================================================================2020-07-05 23:12:51
{'step': 10, 'loss': 0.733, 'accuracy': 0.487}
{'step': 20, 'loss': 0.713, 'accuracy': 0.515}

 +-------+-------+----------+----------+--------------+
| epoch |  loss | accuracy | val_loss | val_accuracy |
+-------+-------+----------+----------+--------------+
|   1   | 0.704 |  0.539   |  0.676   |    0.666     |
+-------+-------+----------+----------+--------------+

==========================================================================================2020-07-05 23:12:51
{'step': 10, 'loss': 0.67, 'accuracy': 0.703}
{'step': 20, 'loss': 0.66, 'accuracy': 0.697}

 +-------+------+----------+----------+--------------+
| epoch | loss | accuracy | val_loss | val_accuracy |
+-------+------+----------+----------+--------------+
|   2   | 0.65 |  0.702   |  0.625   |    0.651     |
+-------+------+----------+----------+--------------+
==========================================================================================2020-07-05 23:13:10
{'step': 10, 'loss': 0.17, 'accuracy': 0.929}
{'step': 20, 'loss': 0.173, 'accuracy': 0.929}

 +-------+-------+----------+----------+--------------+
| epoch |  loss | accuracy | val_loss | val_accuracy |
+-------+-------+----------+----------+--------------+
|   98  | 0.175 |  0.929   |  0.169   |    0.933     |
+-------+-------+----------+----------+--------------+

==========================================================================================2020-07-05 23:13:10
{'step': 10, 'loss': 0.165, 'accuracy': 0.942}
{'step': 20, 'loss': 0.171, 'accuracy': 0.932}

 +-------+-------+----------+----------+--------------+
| epoch |  loss | accuracy | val_loss | val_accuracy |
+-------+-------+----------+----------+--------------+
|   99  | 0.173 |  0.931   |  0.166   |    0.935     |
+-------+-------+----------+----------+--------------+

==========================================================================================2020-07-05 23:13:10
{'step': 10, 'loss': 0.156, 'accuracy': 0.945}
{'step': 20, 'loss': 0.17, 'accuracy': 0.935}

 +-------+-------+----------+----------+--------------+
| epoch |  loss | accuracy | val_loss | val_accuracy |
+-------+-------+----------+----------+--------------+
|  100  | 0.168 |  0.937   |  0.173   |    0.926     |
+-------+-------+----------+----------+--------------+
```

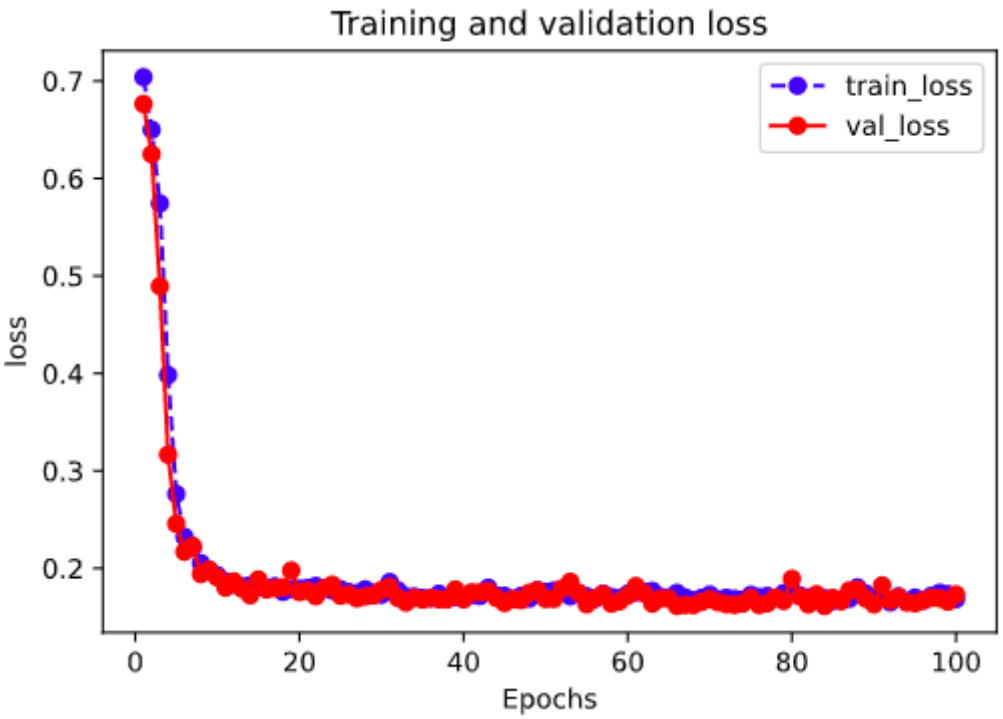
```
==========================================================================================2020-07-05 23:13:11
Finished Training...
```

```
# 结果可视化
fig, (ax1,ax2) = plt.subplots(nrows=1,ncols=2,figsize = (12,5))
ax1.scatter(Xp[:,0],Xp[:,1], c="r")
ax1.scatter(Xn[:,0],Xn[:,1],c = "g")
ax1.legend(["positive","negative"]);
ax1.set_title("y_true");

Xp_pred = X[torch.squeeze(model.forward(X)>=0.5)]
Xn_pred = X[torch.squeeze(model.forward(X)<0.5)]

ax2.scatter(Xp_pred[:,0],Xp_pred[:,1],c = "r")
ax2.scatter(Xn_pred[:,0],Xn_pred[:,1],c = "g")
ax2.legend(["positive","negative"]);
ax2.set_title("y_pred");
```

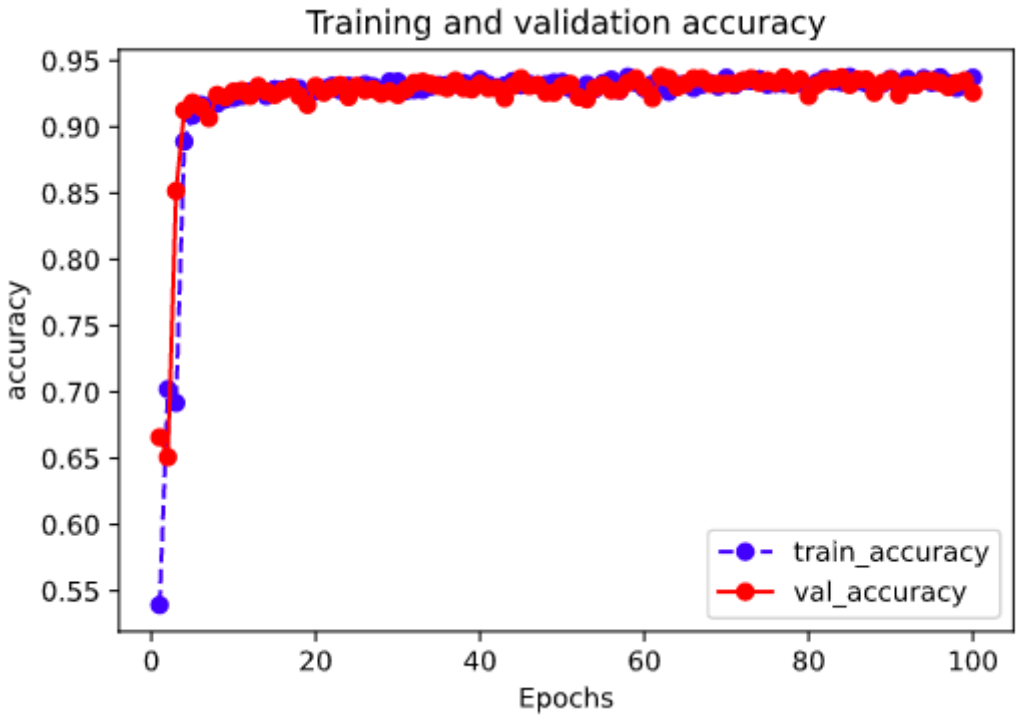


4，评估模型

```
%matplotlib inline
%config InlineBackend.figure_format = 'svg'

import matplotlib.pyplot as plt

def plot_metric(dfhistory, metric):
    train_metrics = dfhistory[metric]
    val_metrics = dfhistory['val_'+metric]
    epochs = range(1, len(train_metrics) + 1)
    plt.plot(epochs, train_metrics, 'bo--')
    plt.plot(epochs, val_metrics, 'ro-')
    plt.title('Training and validation '+ metric)
    plt.xlabel("Epochs")
    plt.ylabel(metric)
    plt.legend(["train_"+metric, 'val_'+metric])
    plt.show()


plot_metric(dfhistory,"loss")
```




```
plot_metric(dfhistory,"accuracy")
```




```
model.evaluate(dl_valid)
```

```
{'val_loss': 0.17309962399303913, 'val_accuracy': 0.9258333394924799}
```

5，使用模型

```
model.predict(dl_valid)[0:10]
```

```
tensor([[0.9998],
        [0.0459],
        [0.0349],
        [0.0147],
        [0.9990],
        [0.9995],
        [0.8535],
        [0.0373],
        [0.2134],
        [0.9356]])
```

4-1,张量的结构操作

张量的操作主要包括张量的结构操作和张量的数学运算。

张量结构操作诸如：张量创建，索引切片，维度变换，合并分割。

张量数学运算主要有：标量运算，向量运算，矩阵运算。另外我们会介绍张量运算的广播机制。

本篇我们介绍张量的结构操作。

一，创建张量

张量创建的许多方法和numpy中创建array的方法很像。

```
import numpy as np
import torch
```

```
a = torch.tensor([1,2,3],dtype = torch.float)
print(a)
```

```
tensor([1., 2., 3.])
```

```
b = torch.arange(1,10,step = 2)
print(b)
```

```
tensor([1, 3, 5, 7, 9])
```

```
c = torch.linspace(0.0,2*3.14,10)
print(c)
```

```
tensor([0.0000, 0.6978, 1.3956, 2.0933, 2.7911, 3.4889, 4.1867, 4.8844, 5.5822,
        6.2800])
```

```
d = torch.zeros((3,3))
print(d)
```

```
tensor([[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]])
```

```
a = torch.ones((3,3),dtype = torch.int)
b = torch.zeros_like(a,dtype = torch.float)
print(a)
print(b)
```

```
tensor([[1, 1, 1],
        [1, 1, 1],
        [1, 1, 1]], dtype=torch.int32)
tensor([[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]])
```

```
torch.fill_(b,5)
print(b)
```

```
tensor([[5., 5., 5.],
        [5., 5., 5.],
        [5., 5., 5.]])
```

```
#均匀随机分布
torch.manual_seed(0)
minval,maxval = 0,10
a = minval + (maxval-minval)*torch.rand([5])
print(a)
```

```
tensor([4.9626, 7.6822, 0.8848, 1.3203, 3.0742])
```

```
#正态分布随机
b = torch.normal(mean = torch.zeros(3,3), std = torch.ones(3,3))
print(b)
```

```
tensor([[-1.3836,  0.2459, -0.1312],
        [-0.1785, -0.5959,  0.2739],
        [ 0.5679, -0.6731, -1.2095]])
```

```
#正态分布随机
mean,std = 2,5
c = std*torch.randn((3,3))+mean
print(c)
```

```
tensor([[  8.7204,  13.9161,  -0.8323],
        [ -3.7681, -10.5115,   6.3778],
        [-11.3628,   1.8433,   4.4939]])
```

```
#整数随机排列
d = torch.randperm(20)
print(d)
```

```
tensor([ 5, 15, 19, 10,  7, 17,  0,  4, 12, 16, 14, 13,  1,  3,  9,  6, 18,  2,
         8, 11])
```

```
#特殊矩阵
I = torch.eye(3,3) #单位矩阵
print(I)
t = torch.diag(torch.tensor([1,2,3])) #对角矩阵
print(t)
```

```
tensor([[1., 0., 0.],
        [0., 1., 0.],
        [0., 0., 1.]])
tensor([[1, 0, 0],
        [0, 2, 0],
        [0, 0, 3]])
```

二，索引切片

张量的索引切片方式和numpy几乎是一样的。切片时支持缺省参数和省略号。

可以通过索引和切片对部分元素进行修改。

此外，对于不规则的切片提取,可以使用torch.index_select, torch.masked_select, torch.take

如果要通过修改张量的某些元素得到新的张量，可以使用torch.where,torch.masked_fill,torch.index_fill

```
#均匀随机分布
torch.manual_seed(0)
minval,maxval = 0,10
t = torch.floor(minval + (maxval-minval)*torch.rand([5,5])).int()
print(t)
```

```
tensor([[4, 7, 0, 1, 3],
        [6, 4, 8, 4, 6],
        [3, 4, 0, 1, 2],
        [5, 6, 8, 1, 2],
        [6, 9, 3, 8, 4]], dtype=torch.int32)
```

```
#第0行
print(t[0])
```

```
tensor([4, 7, 0, 1, 3], dtype=torch.int32)
```

```
#倒数第一行
print(t[-1])
```

```
tensor([6, 9, 3, 8, 4], dtype=torch.int32)
```

```
#第1行第3列
print(t[1,3])
print(t[1][3])
```

```
tensor(4, dtype=torch.int32)
tensor(4, dtype=torch.int32)
```

```
#第1行至第3行
print(t[1:4,:])
```

```
tensor([[6, 4, 8, 4, 6],
        [3, 4, 0, 1, 2],
        [5, 6, 8, 1, 2]], dtype=torch.int32)
```

```
#第1行至最后一行，第0列到最后一列每隔两列取一列
print(t[1:4,:4:2])
```

```
tensor([[6, 8],
        [3, 0],
        [5, 8]], dtype=torch.int32)
```

```
#可以使用索引和切片修改部分元素
x = torch.tensor([[1,2],[3,4]],dtype = torch.float32,requires_grad=True)
x.data[1,:] = torch.tensor([0.0,0.0])
x
```

```
tensor([[1., 2.],
        [0., 0.]], requires_grad=True)
```

```
a = torch.arange(27).view(3,3,3)
print(a)
```

```
tensor([[[ 0,  1,  2],
         [ 3,  4,  5],
         [ 6,  7,  8]],

        [[ 9, 10, 11],
         [12, 13, 14],
         [15, 16, 17]],

        [[18, 19, 20],
         [21, 22, 23],
         [24, 25, 26]]])
```

```
#省略号可以表示多个冒号
print(a[...,1])
```

```
tensor([[ 1,  4,  7],
        [10, 13, 16],
        [19, 22, 25]])
```

以上切片方式相对规则，对于不规则的切片提取,可以使用torch.index_select, torch.take, torch.gather, torch.masked_select.

考虑班级成绩册的例子，有4个班级，每个班级10个学生，每个学生7门科目成绩。可以用一个4×10×7的张量来表示。

```
minval=0
maxval=100
scores = torch.floor(minval + (maxval-minval)*torch.rand([4,10,7])).int()
print(scores)
```

```
tensor([[[55, 95,  3, 18, 37, 30, 93],
         [17, 26, 15,  3, 20, 92, 72],
         [74, 52, 24, 58,  3, 13, 24],
         [81, 79, 27, 48, 81, 99, 69],
         [56, 83, 20, 59, 11, 15, 24],
         [72, 70, 20, 65, 77, 43, 51],
         [61, 81, 98, 11, 31, 69, 91],
         [93, 94, 59,  6, 54, 18,  3],
         [94, 88,  0, 59, 41, 41, 27],
         [69, 20, 68, 75, 85, 68,  0]],

        [[17, 74, 60, 10, 21, 97, 83],
         [28, 37,  2, 49, 12, 11, 47],
         [57, 29, 79, 19, 95, 84,  7],
         [37, 52, 57, 61, 69, 52, 25],
         [73,  2, 20, 37, 25, 32,  9],
         [39, 60, 17, 47, 85, 44, 51],
         [45, 60, 81, 97, 81, 97, 46],
         [ 5, 26, 84, 49, 25, 11,  3],
         [ 7, 39, 77, 77,  1, 81, 10],
         [39, 29, 40, 40,  5,  6, 42]],

        [[50, 27, 68,  4, 46, 93, 29],
         [95, 68,  4, 81, 44, 27, 89],
         [ 9, 55, 39, 85, 63, 74, 67],
         [37, 39,  8, 77, 89, 84, 14],
         [52, 14, 22, 20, 67, 20, 48],
         [52, 82, 12, 15, 20, 84, 32],
         [92, 68, 56, 49, 40, 56, 38],
         [49, 56, 10, 23, 90,  9, 46],
         [99, 68, 51,  6, 74, 14, 35],
         [33, 42, 50, 91, 56, 94, 80]],

        [[18, 72, 14, 28, 64, 66, 87],
         [33, 50, 75,  1, 86,  8, 50],
         [41, 23, 56, 91, 35, 20, 31],
         [ 0, 72, 25, 16, 21, 78, 76],
         [88, 68, 33, 36, 64, 91, 63],
         [26, 26,  2, 60, 21,  5, 93],
         [17, 44, 64, 51, 16,  9, 89],
         [58, 91, 33, 64, 38, 47, 19],
         [66, 65, 48, 38, 19, 84, 12],
         [70, 33, 25, 58, 24, 61, 59]]], dtype=torch.int32)
```

```
#抽取每个班级第0个学生，第5个学生，第9个学生的全部成绩
torch.index_select(scores,dim = 1,index = torch.tensor([0,5,9]))
```

```
tensor([[[55, 95,  3, 18, 37, 30, 93],
         [72, 70, 20, 65, 77, 43, 51],
         [69, 20, 68, 75, 85, 68,  0]],

        [[17, 74, 60, 10, 21, 97, 83],
         [39, 60, 17, 47, 85, 44, 51],
         [39, 29, 40, 40,  5,  6, 42]],

        [[50, 27, 68,  4, 46, 93, 29],
         [52, 82, 12, 15, 20, 84, 32],
         [33, 42, 50, 91, 56, 94, 80]],

        [[18, 72, 14, 28, 64, 66, 87],
         [26, 26,  2, 60, 21,  5, 93],
         [70, 33, 25, 58, 24, 61, 59]]], dtype=torch.int32)
```

```
#抽取每个班级第0个学生，第5个学生，第9个学生的第1门课程，第3门课程，第6门课程成绩
q = torch.index_select(torch.index_select(scores,dim = 1,index = torch.tensor([0,5,9]))
                       ,dim=2,index = torch.tensor([1,3,6]))
print(q)
```

```
tensor([[[95, 18, 93],
         [70, 65, 51],
         [20, 75,  0]],

        [[74, 10, 83],
         [60, 47, 51],
         [29, 40, 42]],

        [[27,  4, 29],
         [82, 15, 32],
         [42, 91, 80]],

        [[72, 28, 87],
         [26, 60, 93],
         [33, 58, 59]]], dtype=torch.int32)
```

```
#抽取第0个班级第0个学生的第0门课程，第2个班级的第4个学生的第1门课程，第3个班级的第9个学生第6门课程成绩
#take将输入看成一维数组，输出和index同形状
s = torch.take(scores,torch.tensor([0*10*7+0,2*10*7+4*7+1,3*10*7+9*7+6]))
s
```

```
<tf.Tensor: shape=(3, 7), dtype=int32, numpy=
array([[52, 82, 66, 55, 17, 86, 14],
       [99, 94, 46, 70,  1, 63, 41],
       [46, 83, 70, 80, 90, 85, 17]], dtype=int32)>
```

```
#抽取分数大于等于80分的分数（布尔索引）
#结果是1维张量
g = torch.masked_select(scores,scores>=80)
print(g)
```

以上这些方法仅能提取张量的部分元素值，但不能更改张量的部分元素值得到新的张量。

如果要通过修改张量的部分元素值得到新的张量，可以使用torch.where,torch.index_fill 和 torch.masked_fill

torch.where可以理解为if的张量版本。

torch.index_fill的选取元素逻辑和torch.index_select相同。

torch.masked_fill的选取元素逻辑和torch.masked_select相同。

```
#如果分数大于60分，赋值成1，否则赋值成0
ifpass = torch.where(scores>60,torch.tensor(1),torch.tensor(0))
print(ifpass)
```

```
#将每个班级第0个学生，第5个学生，第9个学生的全部成绩赋值成满分
torch.index_fill(scores,dim = 1,index = torch.tensor([0,5,9]),value = 100)
#等价于 scores.index_fill(dim = 1,index = torch.tensor([0,5,9]),value = 100)
```

```
#将分数小于60分的分数赋值成60分
b = torch.masked_fill(scores,scores<60,60)
#等价于b = scores.masked_fill(scores<60,60)
b
```

三，维度变换

维度变换相关函数主要有 torch.reshape(或者调用张量的view方法), torch.squeeze, torch.unsqueeze, torch.transpose

torch.reshape 可以改变张量的形状。

torch.squeeze 可以减少维度。

torch.unsqueeze 可以增加维度。

torch.transpose 可以交换维度。

```
# 张量的view方法有时候会调用失败，可以使用reshape方法。

torch.manual_seed(0)
minval,maxval = 0,255
a = (minval + (maxval-minval)*torch.rand([1,3,3,2])).int()
print(a.shape)
print(a)
```

```
torch.Size([1, 3, 3, 2])
tensor([[[[126, 195],
          [ 22,  33],
          [ 78, 161]],

         [[124, 228],
          [116, 161],
          [ 88, 102]],

         [[  5,  43],
          [ 74, 132],
          [177, 204]]]], dtype=torch.int32)
```

```
# 改成  (3,6) 形状的张量
b = a.view([3,6]) #torch.reshape(a,[3,6])
print(b.shape)
print(b)
```

```
torch.Size([3, 6])
tensor([[126, 195,  22,  33,  78, 161],
        [124, 228, 116, 161,  88, 102],
        [  5,  43,  74, 132, 177, 204]], dtype=torch.int32)
```

```
# 改回成 [1,3,3,2] 形状的张量
c = torch.reshape(b,[1,3,3,2]) # b.view([1,3,3,2])
print(c)
```

```
tensor([[[[126, 195],
          [ 22,  33],
          [ 78, 161]],

         [[124, 228],
          [116, 161],
          [ 88, 102]],

         [[  5,  43],
          [ 74, 132],
          [177, 204]]]], dtype=torch.int32)
```

如果张量在某个维度上只有一个元素，利用torch.squeeze可以消除这个维度。

torch.unsqueeze的作用和torch.squeeze的作用相反。

```
a = torch.tensor([[1.0,2.0]])
s = torch.squeeze(a)
print(a)
print(s)
print(a.shape)
print(s.shape)
```

```
tensor([[1., 2.]])
tensor([1., 2.])
torch.Size([1, 2])
torch.Size([2])
```

```
#在第0维插入长度为1的一个维度

d = torch.unsqueeze(s,axis=0)
print(s)
print(d)

print(s.shape)
print(d.shape)
```

```
tensor([1., 2.])
tensor([[1., 2.]])
torch.Size([2])
torch.Size([1, 2])
```

torch.transpose可以交换张量的维度，torch.transpose常用于图片存储格式的变换上。

如果是二维的矩阵，通常会调用矩阵的转置方法 matrix.t()，等价于 torch.transpose(matrix,0,1)。

```
minval=0
maxval=255
# Batch,Height,Width,Channel
data = torch.floor(minval + (maxval-minval)*torch.rand([100,256,256,4])).int()
print(data.shape)

# 转换成 Pytorch默认的图片格式 Batch,Channel,Height,Width
# 需要交换两次
data_t = torch.transpose(torch.transpose(data,1,2),1,3)
print(data_t.shape)
```

```
torch.Size([100, 256, 256, 4])
torch.Size([100, 4, 256, 256])
```

```
matrix = torch.tensor([[1,2,3],[4,5,6]])
print(matrix)
print(matrix.t()) #等价于torch.transpose(matrix,0,1)
```

```
tensor([[1, 2, 3],
        [4, 5, 6]])
tensor([[1, 4],
        [2, 5],
        [3, 6]])
```

四，合并分割

可以用torch.cat方法和torch.stack方法将多个张量合并，可以用torch.split方法把一个张量分割成多个张量。

torch.cat和torch.stack有略微的区别，torch.cat是连接，不会增加维度，而torch.stack是堆叠，会增加维度。

```
a = torch.tensor([[1.0,2.0],[3.0,4.0]])
b = torch.tensor([[5.0,6.0],[7.0,8.0]])
c = torch.tensor([[9.0,10.0],[11.0,12.0]])

abc_cat = torch.cat([a,b,c],dim = 0)
print(abc_cat.shape)
print(abc_cat)
```

```
torch.Size([6, 2])
tensor([[ 1.,  2.],
        [ 3.,  4.],
        [ 5.,  6.],
        [ 7.,  8.],
        [ 9., 10.],
        [11., 12.]])
```

```
abc_stack = torch.stack([a,b,c],axis = 0) #torch中dim和axis参数名可以混用
print(abc_stack.shape)
print(abc_stack)
```

```
torch.Size([3, 2, 2])
tensor([[[ 1.,  2.],
         [ 3.,  4.]],

        [[ 5.,  6.],
         [ 7.,  8.]],

        [[ 9., 10.],
         [11., 12.]]])
```

```
torch.cat([a,b,c],axis = 1)
```

```
tensor([[ 1.,  2.,  5.,  6.,  9., 10.],
        [ 3.,  4.,  7.,  8., 11., 12.]])
```

```
torch.stack([a,b,c],axis = 1)
```

```
tensor([[[ 1.,  2.],
         [ 5.,  6.],
         [ 9., 10.]],

        [[ 3.,  4.],
         [ 7.,  8.],
         [11., 12.]]])
```

torch.split是torch.cat的逆运算，可以指定分割份数平均分割，也可以通过指定每份的记录数量进行分割。

```
print(abc_cat)
a,b,c = torch.split(abc_cat,split_size_or_sections = 2,dim = 0) #每份2个进行分割
print(a)
print(b)
print(c)
```

```
print(abc_cat)
p,q,r = torch.split(abc_cat,split_size_or_sections =[4,1,1],dim = 0) #每份分别为[4,1,1]
print(p)
print(q)
print(r)
```

```
tensor([[ 1.,  2.],
        [ 3.,  4.],
        [ 5.,  6.],
        [ 7.,  8.],
        [ 9., 10.],
        [11., 12.]])
tensor([[1., 2.],
        [3., 4.],
        [5., 6.],
        [7., 8.]])
tensor([[ 9., 10.]])
tensor([[11., 12.]])
```

4-2,张量的数学运算

张量的操作主要包括张量的结构操作和张量的数学运算。

张量结构操作诸如：张量创建，索引切片，维度变换，合并分割。

张量数学运算主要有：标量运算，向量运算，矩阵运算。另外我们会介绍张量运算的广播机制。

本篇我们介绍张量的数学运算。

本篇文章部分内容参考如下博客：https://blog.csdn.net/duan_zhihua/article/details/82526505

一，标量运算

张量的数学运算符可以分为标量运算符、向量运算符、以及矩阵运算符。

加减乘除乘方，以及三角函数，指数，对数等常见函数，逻辑比较运算符等都是标量运算符。

标量运算符的特点是对张量实施逐元素运算。

有些标量运算符对常用的数学运算符进行了重载。并且支持类似numpy的广播特性。

```
import torch
import numpy as np
```

```
a = torch.tensor([[1.0,2],[-3,4.0]])
b = torch.tensor([[5.0,6],[7.0,8.0]])
a+b  #运算符重载
```

```
tensor([[ 6.,  8.],
        [ 4., 12.]])
```

```
a-b
```

```
tensor([[ -4.,  -4.],
        [-10.,  -4.]])
```

```
a*b
```

```
tensor([[  5.,  12.],
        [-21.,  32.]])
```

```
a/b
```

```
tensor([[ 0.2000,  0.3333],
        [-0.4286,  0.5000]])
```

```
a**2
```

```
tensor([[ 1.,  4.],
        [ 9., 16.]])
```

```
a**(0.5)
```

```
tensor([[1.0000, 1.4142],
        [   nan, 2.0000]])
```

```
a%3 #求模
```

```
tensor([[1., 2.],
        [0., 1.]])
```

```
a//3  #地板除法
```

```
tensor([[ 0.,  0.],
        [-1.,  1.]])
```

```
a>=2 # torch.ge(a,2)  #ge: greater_equal缩写
```

```
tensor([[False,  True],
        [False,  True]])
```

```
(a>=2)&(a<=3)
```

```
tensor([[False,  True],
        [False, False]])
```

```
(a>=2)|(a<=3)
```

```
tensor([[True, True],
        [True, True]])
```

```
a==5 #torch.eq(a,5)
```

```
tensor([[False, False],
        [False, False]])
```

```
torch.sqrt(a)
```

```
tensor([[1.0000, 1.4142],
        [   nan, 2.0000]])
```

```
a = torch.tensor([1.0,8.0])
b = torch.tensor([5.0,6.0])
c = torch.tensor([6.0,7.0])

d = a+b+c
print(d)
```

```
tensor([12., 21.])
```

```
print(torch.max(a,b))
```

```
tensor([5., 8.])
```

```
print(torch.min(a,b))
```

```
tensor([1., 6.])
```

```
x = torch.tensor([2.6,-2.7])

print(torch.round(x)) #保留整数部分，四舍五入
print(torch.floor(x)) #保留整数部分，向下归整
print(torch.ceil(x))  #保留整数部分，向上归整
print(torch.trunc(x)) #保留整数部分，向0归整
```

```
tensor([ 3., -3.])
tensor([ 2., -3.])
tensor([ 3., -2.])
tensor([ 2., -2.])
```

```
x = torch.tensor([2.6,-2.7])
print(torch.fmod(x,2)) #作除法取余数
print(torch.remainder(x,2)) #作除法取剩余的部分，结果恒正
```

```
tensor([ 0.6000, -0.7000])
tensor([0.6000, 1.3000])
```

```
# 幅值裁剪
x = torch.tensor([0.9,-0.8,100.0,-20.0,0.7])
y = torch.clamp(x,min=-1,max = 1)
z = torch.clamp(x,max = 1)
print(y)
print(z)
```

```
tensor([ 0.9000, -0.8000,  1.0000, -1.0000,  0.7000])
tensor([  0.9000,  -0.8000,   1.0000, -20.0000,   0.7000])
```

二，向量运算

向量运算符只在一个特定轴上运算，将一个向量映射到一个标量或者另外一个向量。

```
#统计值

a = torch.arange(1,10).float()
print(torch.sum(a))
print(torch.mean(a))
print(torch.max(a))
print(torch.min(a))
print(torch.prod(a)) #累乘
print(torch.std(a))  #标准差
print(torch.var(a))  #方差
print(torch.median(a)) #中位数
```

```
tensor(45.)
tensor(5.)
tensor(9.)
tensor(1.)
tensor(362880.)
tensor(2.7386)
tensor(7.5000)
tensor(5.)
```

#指定维度计算统计值

```
b = a.view(3,3)
print(b)
print(torch.max(b,dim = 0))
print(torch.max(b,dim = 1))
```

```
tensor([[1., 2., 3.],
        [4., 5., 6.],
        [7., 8., 9.]])
torch.return_types.max(
values=tensor([7., 8., 9.]),
indices=tensor([2, 2, 2]))
torch.return_types.max(
values=tensor([3., 6., 9.]),
indices=tensor([2, 2, 2]))
```

#cum扫描

```
a = torch.arange(1,10)

print(torch.cumsum(a,0))
print(torch.cumprod(a,0))
print(torch.cummax(a,0).values)
print(torch.cummax(a,0).indices)
print(torch.cummin(a,0))
```

```
tensor([ 1,  3,  6, 10, 15, 21, 28, 36, 45])
tensor([     1,      2,      6,     24,    120,    720,   5040,  40320, 362880])
tensor([1, 2, 3, 4, 5, 6, 7, 8, 9])
tensor([0, 1, 2, 3, 4, 5, 6, 7, 8])
torch.return_types.cummin(
values=tensor([1, 1, 1, 1, 1, 1, 1, 1, 1]),
indices=tensor([0, 0, 0, 0, 0, 0, 0, 0, 0]))
```

```
#torch.sort和torch.topk可以对张量排序
a = torch.tensor([[9,7,8],[1,3,2],[5,6,4]]).float()
print(torch.topk(a,2,dim = 0),"\n")
print(torch.topk(a,2,dim = 1),"\n")
print(torch.sort(a,dim = 1),"\n")

#利用torch.topk可以在Pytorch中实现KNN算法
```

```
torch.return_types.topk(
values=tensor([[9., 7., 8.],
        [5., 6., 4.]]),
indices=tensor([[0, 0, 0],
        [2, 2, 2]]))

torch.return_types.topk(
values=tensor([[9., 8.],
        [3., 2.],
        [6., 5.]]),
indices=tensor([[0, 2],
        [1, 2],
        [1, 0]]))

torch.return_types.sort(
values=tensor([[7., 8., 9.],
        [1., 2., 3.],
        [4., 5., 6.]]),
indices=tensor([[1, 2, 0],
        [0, 2, 1],
        [2, 0, 1]]))
```

三，矩阵运算

矩阵必须是二维的。类似torch.tensor([1,2,3])这样的不是矩阵。

矩阵运算包括：矩阵乘法，矩阵转置，矩阵逆，矩阵求迹，矩阵范数，矩阵行列式，矩阵求特征值，矩阵分解等运算。

```
#矩阵乘法
a = torch.tensor([[1,2],[3,4]])
b = torch.tensor([[2,0],[0,2]])
print(a@b)  #等价于torch.matmul(a,b) 或 torch.mm(a,b)
```

```
tensor([[2, 4],
        [6, 8]])
```

```
#矩阵转置
a = torch.tensor([[1.0,2],[3,4]])
print(a.t())
```

```
tensor([[1., 3.],
        [2., 4.]])
```

```
#矩阵逆，必须为浮点类型
a = torch.tensor([[1.0,2],[3,4]])
print(torch.inverse(a))
```

```
tensor([[-2.0000,  1.0000],
        [ 1.5000, -0.5000]])
```

```
#矩阵求trace
a = torch.tensor([[1.0,2],[3,4]])
print(torch.trace(a))
```

```
tensor(5.)
```

```
#矩阵求范数
a = torch.tensor([[1.0,2],[3,4]])
print(torch.norm(a))
```

```
tensor(5.4772)
```

```
#矩阵行列式
a = torch.tensor([[1.0,2],[3,4]])
print(torch.det(a))
```

```
tensor(-2.0000)
```

```
#矩阵特征值和特征向量
a = torch.tensor([[1.0,2],[-5,4]],dtype = torch.float)
print(torch.eig(a,eigenvectors=True))

#两个特征值分别是 -2.5+2.7839j, 2.5-2.7839j


torch.return_types.eig(
eigenvalues=tensor([[ 2.5000,  2.7839],
        [ 2.5000, -2.7839]]),
eigenvectors=tensor([[ 0.2535, -0.4706],
        [ 0.8452,  0.0000]]))


#矩阵QR分解，将一个方阵分解为一个正交矩阵q和上三角矩阵r
#QR分解实际上是对矩阵a实施Schmidt正交化得到q

a  = torch.tensor([[1.0,2.0],[3.0,4.0]])
q,r = torch.qr(a)
print(q,"\n")
print(r,"\n")
print(q@r)




#矩阵svd分解
#svd分解可以将任意一个矩阵分解为一个正交矩阵u,一个对角阵s和一个正交矩阵v.t()的乘积
#svd常用于矩阵压缩和降维
a=torch.tensor([[1.0,2.0],[3.0,4.0],[5.0,6.0]])

u,s,v = torch.svd(a)

print(u,"\n")
print(s,"\n")
print(v,"\n")

print(u@torch.diag(s)@v.t())

#利用svd分解可以在Pytorch中实现主成分分析降维
```

```
tensor([[-0.2298,  0.8835],
        [-0.5247,  0.2408],
        [-0.8196, -0.4019]])

tensor([9.5255, 0.5143])

tensor([[-0.6196, -0.7849],
        [-0.7849,  0.6196]])

tensor([[1.0000, 2.0000],
        [3.0000, 4.0000],
        [5.0000, 6.0000]])
```

四，广播机制

Pytorch的广播规则和numpy是一样的:

- 1、如果张量的维度不同，将维度较小的张量进行扩展，直到两个张量的维度都一样。
- 2、如果两个张量在某个维度上的长度是相同的，或者其中一个张量在该维度上的长度为1，那么我们就说这两个张量在该维度上是相容的。
- 3、如果两个张量在所有维度上都是相容的，它们就能使用广播。
- 4、广播之后，每个维度的长度将取两个张量在该维度长度的较大值。
- 5、在任何一个维度上，如果一个张量的长度为1，另一个张量长度大于1，那么在该维度上，就好像是对第一个张量进行了复制。

torch.broadcast_tensors可以将多个张量根据广播规则转换成相同的维度。

```
a = torch.tensor([1,2,3])
b = torch.tensor([[0,0,0],[1,1,1],[2,2,2]])
print(b + a)
```

```
tensor([[1, 2, 3],
        [2, 3, 4],
        [3, 4, 5]])
```

```
a_broad,b_broad = torch.broadcast_tensors(a,b)
print(a_broad,"\n")
print(b_broad,"\n")
print(a_broad + b_broad)
```

```
tensor([[1, 2, 3],
        [1, 2, 3],
        [1, 2, 3]])

tensor([[0, 0, 0],
        [1, 1, 1],
        [2, 2, 2]])

tensor([[1, 2, 3],
        [2, 3, 4],
        [3, 4, 5]])
```

如果对本书内容理解上有需要进一步和作者交流的地方，欢迎在公众号"Python与算法之美"下留言。作者时间和精力有限，会酌情予以回复。

也可以在公众号后台回复关键字：**加群**，加入读者交流群和大家讨论。

4-3,nn.functional 和 nn.Module

```
import os
import datetime

#打印时间
def printbar():
    nowtime = datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S')
    print("\n"+"=========="*8 + "%s"%nowtime)

#mac系统上pytorch和matplotlib在jupyter中同时跑需要更改环境变量
os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"
```

一，nn.functional 和 nn.Module

前面我们介绍了Pytorch的张量的结构操作和数学运算中的一些常用API。

利用这些张量的API我们可以构建出神经网络相关的组件(如激活函数，模型层，损失函数)。

Pytorch和神经网络相关的功能组件大多都封装在 torch.nn模块下。

这些功能组件的绝大部分既有函数形式实现，也有类形式实现。

其中nn.functional(一般引入后改名为F)有各种功能组件的函数实现。例如：

(激活函数)

- F.relu
- F.sigmoid
- F.tanh
- F.softmax

(模型层)

- F.linear
- F.conv2d
- F.max_pool2d
- F.dropout2d
- F.embedding

(损失函数)

- F.binary_cross_entropy
- F.mse_loss
- F.cross_entropy

为了便于对参数进行管理，一般通过继承 nn.Module 转换成为类的实现形式，并直接封装在 nn 模块下。例如：

(激活函数)

- nn.ReLU
- nn.Sigmoid
- nn.Tanh
- nn.Softmax

(模型层)

- nn.Linear
- nn.Conv2d
- nn.MaxPool2d
- nn.Dropout2d
- nn.Embedding

(损失函数)

- nn.BCELoss
- nn.MSELoss
- nn.CrossEntropyLoss

实际上nn.Module除了可以管理其引用的各种参数，还可以管理其引用的子模块，功能十分强大。

二，使用nn.Module来管理参数

在Pytorch中，模型的参数是需要被优化器训练的，因此，通常要设置参数为 requires_grad = True 的张量。

同时，在一个模型中，往往有许多的参数，要手动管理这些参数并不是一件容易的事情。

Pytorch一般将参数用nn.Parameter来表示，并且用nn.Module来管理其结构下的所有参数。

```
import torch
from torch import nn
import torch.nn.functional  as F
from matplotlib import pyplot as plt
```

```
# nn.Parameter 具有 requires_grad = True 属性
w = nn.Parameter(torch.randn(2,2))
print(w)
print(w.requires_grad)
```

```
Parameter containing:
tensor([[ 0.3544, -1.1643],
        [ 1.2302,  1.3952]], requires_grad=True)
True
```

```
# nn.ParameterList 可以将多个nn.Parameter组成一个列表
params_list = nn.ParameterList([nn.Parameter(torch.rand(8,i)) for i in range(1,3)])
print(params_list)
print(params_list[0].requires_grad)
```

```
ParameterList(
    (0): Parameter containing: [torch.FloatTensor of size 8x1]
    (1): Parameter containing: [torch.FloatTensor of size 8x2]
)
True
```

```
# nn.ParameterDict 可以将多个nn.Parameter组成一个字典

params_dict = nn.ParameterDict({"a":nn.Parameter(torch.rand(2,2)),
                                "b":nn.Parameter(torch.zeros(2))})
print(params_dict)
print(params_dict["a"].requires_grad)
```

```
ParameterDict(
    (a): Parameter containing: [torch.FloatTensor of size 2x2]
    (b): Parameter containing: [torch.FloatTensor of size 2]
)
True
```

```
# 可以用Module将它们管理起来
# module.parameters()返回一个生成器，包括其结构下的所有parameters

module = nn.Module()
module.w = w
module.params_list = params_list
module.params_dict = params_dict

num_param = 0
for param in module.parameters():
    print(param,"\n")
    num_param = num_param + 1
print("number of Parameters =",num_param)
```

```
Parameter containing:
tensor([[ 0.3544, -1.1643],
        [ 1.2302,  1.3952]], requires_grad=True)

Parameter containing:
tensor([[0.9391],
        [0.7590],
        [0.6899],
        [0.4786],
        [0.2392],
        [0.9645],
        [0.1968],
        [0.1353]], requires_grad=True)

Parameter containing:
tensor([[0.8012, 0.9587],
        [0.0276, 0.5995],
        [0.7338, 0.5559],
        [0.1704, 0.5814],
        [0.7626, 0.1179],
        [0.4945, 0.2408],
        [0.7179, 0.0575],
        [0.3418, 0.7291]], requires_grad=True)

Parameter containing:
tensor([[0.7729, 0.2383],
        [0.7054, 0.9937]], requires_grad=True)

Parameter containing:
tensor([0., 0.], requires_grad=True)

number of Parameters = 5
```

```
#实践当中，一般通过继承nn.Module来构建模块类，并将所有含有需要学习的参数的部分放在构造函数中。

#以下范例为Pytorch中nn.Linear的源码的简化版本
#可以看到它将需要学习的参数放在了__init__构造函数中，并在forward中调用F.linear函数来实现计算逻辑。

class Linear(nn.Module):
    __constants__ = ['in_features', 'out_features']

    def __init__(self, in_features, out_features, bias=True):
        super(Linear, self).__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.weight = nn.Parameter(torch.Tensor(out_features, in_features))
        if bias:
            self.bias = nn.Parameter(torch.Tensor(out_features))
        else:
            self.register_parameter('bias', None)

    def forward(self, input):
        return F.linear(input, self.weight, self.bias)
```

三，使用nn.Module来管理子模块

一般情况下，我们都很少直接使用 nn.Parameter来定义参数构建模型，而是通过一些拼装一些常用的模型层来构造模型。

这些模型层也是继承自nn.Module的对象,本身也包括参数，属于我们要定义的模块的子模块。

nn.Module提供了一些方法可以管理这些子模块。

- children() 方法: 返回生成器，包括模块下的所有子模块。
- named_children()方法：返回一个生成器，包括模块下的所有子模块，以及它们的名字。
- modules()方法：返回一个生成器，包括模块下的所有各个层级的模块，包括模块本身。
- named_modules()方法：返回一个生成器，包括模块下的所有各个层级的模块以及它们的名字，包括模块本身。

其中chidren()方法和named_children()方法较多使用。

modules()方法和named_modules()方法较少使用，其功能可以通过多个named_children()的嵌套使用实现。

```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()

        self.embedding = nn.Embedding(num_embeddings = 10000,embedding_dim = 3,padding_idx = 1)
        self.conv = nn.Sequential()
        self.conv.add_module("conv_1",nn.Conv1d(in_channels = 3,out_channels = 16,kernel_size =
        self.conv.add_module("pool_1",nn.MaxPool1d(kernel_size = 2))
        self.conv.add_module("relu_1",nn.ReLU())
        self.conv.add_module("conv_2",nn.Conv1d(in_channels = 16,out_channels = 128,kernel_size
        self.conv.add_module("pool_2",nn.MaxPool1d(kernel_size = 2))
        self.conv.add_module("relu_2",nn.ReLU())

        self.dense = nn.Sequential()
        self.dense.add_module("flatten",nn.Flatten())
        self.dense.add_module("linear",nn.Linear(6144,1))
        self.dense.add_module("sigmoid",nn.Sigmoid())

    def forward(self,x):
        x = self.embedding(x).transpose(1,2)
        x = self.conv(x)
        y = self.dense(x)
        return y

net = Net()



i = 0
for child in net.children():
    i+=1
    print(child,"\n")
print("child number",i)
```

```
Embedding(10000, 3, padding_idx=1)

Sequential(
  (conv_1): Conv1d(3, 16, kernel_size=(5,), stride=(1,))
  (pool_1): MaxPool1d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (relu_1): ReLU()
  (conv_2): Conv1d(16, 128, kernel_size=(2,), stride=(1,))
  (pool_2): MaxPool1d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (relu_2): ReLU()
)

Sequential(
  (flatten): Flatten()
  (linear): Linear(in_features=6144, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)

child number 3
```

```
i = 0
for name,child in net.named_children():
    i+=1
    print(name,":",child,"\n")
print("child number",i)
```

```
embedding : Embedding(10000, 3, padding_idx=1)

conv : Sequential(
  (conv_1): Conv1d(3, 16, kernel_size=(5,), stride=(1,))
  (pool_1): MaxPool1d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (relu_1): ReLU()
  (conv_2): Conv1d(16, 128, kernel_size=(2,), stride=(1,))
  (pool_2): MaxPool1d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (relu_2): ReLU()
)

dense : Sequential(
  (flatten): Flatten()
  (linear): Linear(in_features=6144, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)

child number 3
```

```
i = 0
for module in net.modules():
    i+=1
    print(module)
print("module number:",i)
```

```
Net(
  (embedding): Embedding(10000, 3, padding_idx=1)
  (conv): Sequential(
    (conv_1): Conv1d(3, 16, kernel_size=(5,), stride=(1,))
    (pool_1): MaxPool1d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (relu_1): ReLU()
    (conv_2): Conv1d(16, 128, kernel_size=(2,), stride=(1,))
    (pool_2): MaxPool1d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (relu_2): ReLU()
  )
  (dense): Sequential(
    (flatten): Flatten()
    (linear): Linear(in_features=6144, out_features=1, bias=True)
    (sigmoid): Sigmoid()
  )
)
Embedding(10000, 3, padding_idx=1)
Sequential(
  (conv_1): Conv1d(3, 16, kernel_size=(5,), stride=(1,))
  (pool_1): MaxPool1d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (relu_1): ReLU()
  (conv_2): Conv1d(16, 128, kernel_size=(2,), stride=(1,))
  (pool_2): MaxPool1d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (relu_2): ReLU()
)
Conv1d(3, 16, kernel_size=(5,), stride=(1,))
MaxPool1d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
ReLU()
Conv1d(16, 128, kernel_size=(2,), stride=(1,))
MaxPool1d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
ReLU()
Sequential(
  (flatten): Flatten()
  (linear): Linear(in_features=6144, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)
Flatten()
Linear(in_features=6144, out_features=1, bias=True)
Sigmoid()
module number: 13
```

下面我们通过named_children方法找到embedding层，并将其参数设置为不可训练(相当于冻结embedding层)。

```
children_dict = {name:module for name,module in net.named_children()}

print(children_dict)
embedding = children_dict["embedding"]
embedding.requires_grad_(False) #冻结其参数
```

```
{'embedding': Embedding(10000, 3, padding_idx=1), 'conv': Sequential(
  (conv_1): Conv1d(3, 16, kernel_size=(5,), stride=(1,))
  (pool_1): MaxPool1d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (relu_1): ReLU()
  (conv_2): Conv1d(16, 128, kernel_size=(2,), stride=(1,))
  (pool_2): MaxPool1d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (relu_2): ReLU()
), 'dense': Sequential(
  (flatten): Flatten()
  (linear): Linear(in_features=6144, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)}
```

```
#可以看到其第一层的参数已经不可以被训练了。
for param in embedding.parameters():
    print(param.requires_grad)
    print(param.numel())
```

```
False
30000
```

```
from torchkeras import summary
summary(net,input_shape = (200,),input_dtype = torch.LongTensor)
# 不可训练参数数量增加
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
         Embedding-1               [-1, 200, 3]          30,000
            Conv1d-2              [-1, 16, 196]             256
         MaxPool1d-3               [-1, 16, 98]               0
              ReLU-4               [-1, 16, 98]               0
            Conv1d-5              [-1, 128, 97]           4,224
         MaxPool1d-6              [-1, 128, 48]               0
              ReLU-7              [-1, 128, 48]               0
           Flatten-8                  [-1, 6144]               0
            Linear-9                     [-1, 1]           6,145
          Sigmoid-10                     [-1, 1]               0
================================================================
Total params: 40,625
Trainable params: 10,625
Non-trainable params: 30,000
----------------------------------------------------------------
Input size (MB): 0.000763
Forward/backward pass size (MB): 0.287796
Params size (MB): 0.154972
Estimated Total Size (MB): 0.443531
----------------------------------------------------------------
```

如果对本书内容理解上有需要进一步和作者交流的地方，欢迎在公众号"Python与算法之美"下留言。作者时间和精力有限，会酌情予以回复。

也可以在公众号后台回复关键字：**加群**，加入读者交流群和大家讨论。

5-1, Dataset和DataLoader

Pytorch通常使用Dataset和DataLoader这两个工具类来构建数据管道。

Dataset定义了数据集的内容，它相当于一个类似列表的数据结构，具有确定的长度，能够用索引获取数据集中的元素。

而DataLoader定义了按batch加载数据集的方法，它是一个实现了 `__iter__` 方法的可迭代对象，每次迭代输出一个batch的数据。

DataLoader能够控制batch的大小，batch中元素的采样方法，以及将batch结果整理成模型所需输入形式的方法，并且能够使用多进程读取数据。

在绝大部分情况下，用户只需实现Dataset的 `__len__` 方法和 `__getitem__` 方法，就可以轻松构建自己的数据集，并用默认数据管道进行加载。

一，Dataset和DataLoader概述

1，获取一个batch数据的步骤

让我们考虑一下从一个数据集中获取一个batch的数据需要哪些步骤。

(假定数据集的特征和标签分别表示为张量 `X` 和 `Y` ，数据集可以表示为 `(X,Y)` , 假定batch大小为 `m`)

1，首先我们要确定数据集的长度 `n` 。

结果类似： `n = 1000` 。

2，然后我们从 `0` 到 `n-1` 的范围中抽样出 `m` 个数(batch大小)。

假定 `m=4` , 拿到的结果是一个列表，类似： `indices = [1,4,8,9]`

3，接着我们从数据集中去取这 `m` 个数对应下标的元素。

拿到的结果是一个元组列表，类似： `samples = [(X[1],Y[1]),(X[4],Y[4]),(X[8],Y[8]),(X[9],Y[9])]`

4，最后我们将结果整理成两个张量作为输出。

拿到的结果是两个张量，类似 `batch = (features,labels)` ,

其中 `features = torch.stack([X[1],X[4],X[8],X[9]])`

```
labels = torch.stack([Y[1],Y[4],Y[8],Y[9]])
```

2，Dataset和DataLoader的功能分工

上述第1个步骤确定数据集的长度是由 Dataset的 `__len__` 方法实现的。

第2个步骤从 `0` 到 `n-1` 的范围中抽样出 `m` 个数的方法是由 DataLoader的 `sampler` 和 `batch_sampler` 参数指定的。

`sampler` 参数指定单个元素抽样方法，一般无需用户设置，程序默认在DataLoader的参数 `shuffle=True` 时采用随机抽样， `shuffle=False` 时采用顺序抽样。

`batch_sampler` 参数将多个抽样的元素整理成一个列表，一般无需用户设置，默认方法在DataLoader的参数 `drop_last=True` 时会丢弃数据集最后一个长度不能被batch大小整除的批次，在 `drop_last=False` 时保留最后一个批次。

第3个步骤的核心逻辑根据下标取数据集中的元素 是由 Dataset的 `__getitem__` 方法实现的。

第4个步骤的逻辑由DataLoader的参数 `collate_fn` 指定。一般情况下也无需用户设置。

3，Dataset和DataLoader的主要接口

以下是 Dataset和 DataLoader的核心接口逻辑伪代码，不完全和源码一致。

```
import torch
class Dataset(object):
    def __init__(self):
        pass

    def __len__(self):
        raise NotImplementedError

    def __getitem__(self,index):
        raise NotImplementedError


class DataLoader(object):
    def __init__(self,dataset,batch_size,collate_fn,shuffle = True,drop_last = False):
        self.dataset = dataset
        self.sampler =torch.utils.data.RandomSampler if shuffle else \
           torch.utils.data.SequentialSampler
        self.batch_sampler = torch.utils.data.BatchSampler
        self.sample_iter = self.batch_sampler(
            self.sampler(range(len(dataset))),
            batch_size = batch_size,drop_last = drop_last)

    def __next__(self):
        indices = next(self.sample_iter)
        batch = self.collate_fn([self.dataset[i] for i in indices])
        return batch
```

二，使用Dataset创建数据集

Dataset创建数据集常用的方法有：

- 使用 torch.utils.data.TensorDataset 根据Tensor创建数据集(numpy的array，Pandas的DataFrame需要先转换成Tensor)。
- 使用 torchvision.datasets.ImageFolder 根据图片目录创建图片数据集。
- 继承 torch.utils.data.Dataset 创建自定义数据集。

此外，还可以通过

- torch.utils.data.random_split 将一个数据集分割成多份，常用于分割训练集，验证集和测试集。
- 调用Dataset的加法运算符(+)将多个数据集合并成一个数据集。

1，根据Tensor创建数据集

```
import numpy as np
import torch
from torch.utils.data import TensorDataset,Dataset,DataLoader,random_split



# 根据Tensor创建数据集

from sklearn import datasets
iris = datasets.load_iris()
ds_iris = TensorDataset(torch.tensor(iris.data),torch.tensor(iris.target))

# 分割成训练集和预测集
n_train = int(len(ds_iris)*0.8)
n_valid = len(ds_iris) - n_train
ds_train,ds_valid = random_split(ds_iris,[n_train,n_valid])

print(type(ds_iris))
print(type(ds_train))



# 使用DataLoader加载数据集
dl_train,dl_valid = DataLoader(ds_train,batch_size = 8),DataLoader(ds_valid,batch_size = 8)

for features,labels in dl_train:
    print(features,labels)
    break


# 演示加法运算符（`+`）的合并作用

ds_data = ds_train + ds_valid

print('len(ds_train) = ',len(ds_train))
print('len(ds_valid) = ',len(ds_valid))
print('len(ds_train+ds_valid) = ',len(ds_data))

print(type(ds_data))
```

2，根据图片目录创建图片数据集

```
import numpy as np
import torch
from torch.utils.data import DataLoader
from torchvision import transforms,datasets
```

#演示一些常用的图片增强操作

```
from PIL import Image
img = Image.open('./data/cat.jpeg')
img
```


```
# 随机数值翻转
transforms.RandomVerticalFlip()(img)
```

```
#随机旋转
transforms.RandomRotation(45)(img)
```


```
# 定义图片增强操作

transform_train = transforms.Compose([
   transforms.RandomHorizontalFlip(), #随机水平翻转
   transforms.RandomVerticalFlip(), #随机垂直翻转
   transforms.RandomRotation(45),  #随机在45度角度内旋转
   transforms.ToTensor() #转换成张量
  ]
)

transform_valid = transforms.Compose([
    transforms.ToTensor()
  ]
)
```

```
# 根据图片目录创建数据集
ds_train = datasets.ImageFolder("./data/cifar2/train/",
            transform = transform_train,target_transform= lambda t:torch.tensor([t]).float())
ds_valid = datasets.ImageFolder("./data/cifar2/test/",
            transform = transform_train,target_transform= lambda t:torch.tensor([t]).float())

print(ds_train.class_to_idx)
```

```
{'0_airplane': 0, '1_automobile': 1}
```

```
# 使用DataLoader加载数据集

dl_train = DataLoader(ds_train,batch_size = 50,shuffle = True,num_workers=3)
dl_valid = DataLoader(ds_valid,batch_size = 50,shuffle = True,num_workers=3)
```

```
for features,labels in dl_train:
    print(features.shape)
    print(labels.shape)
    break
```

```
torch.Size([50, 3, 32, 32])
torch.Size([50, 1])
```

3，创建自定义数据集

下面通过继承Dataset类创建imdb文本分类任务的自定义数据集。

大概思路如下：首先，对训练集文本分词构建词典。然后将训练集文本和测试集文本数据转换成token单词编码。

接着将转换成单词编码的训练集数据和测试集数据按样本分割成多个文件，一个文件代表一个样本。

最后，我们可以根据文件名列表获取对应序号的样本内容，从而构建Dataset数据集。

```
import numpy as np
import pandas as pd
from collections import OrderedDict
import re,string

MAX_WORDS = 10000  # 仅考虑最高频的10000个词
MAX_LEN = 200  # 每个样本保留200个词的长度
BATCH_SIZE = 20

train_data_path = 'data/imdb/train.tsv'
test_data_path = 'data/imdb/test.tsv'
train_token_path = 'data/imdb/train_token.tsv'
test_token_path =  'data/imdb/test_token.tsv'
train_samples_path = 'data/imdb/train_samples/'
test_samples_path =  'data/imdb/test_samples/'
```

首先我们构建词典，并保留最高频的MAX_WORDS个词。

```
##构建词典

word_count_dict = {}

#清洗文本
def clean_text(text):
    lowercase = text.lower().replace("\n"," ")
    stripped_html = re.sub('<br />', ' ',lowercase)
    cleaned_punctuation = re.sub('[%s]'%re.escape(string.punctuation),'',stripped_html)
    return cleaned_punctuation

with open(train_data_path,"r",encoding = 'utf-8') as f:
    for line in f:
        label,text = line.split("\t")
        cleaned_text = clean_text(text)
        for word in cleaned_text.split(" "):
            word_count_dict[word] = word_count_dict.get(word,0)+1

df_word_dict = pd.DataFrame(pd.Series(word_count_dict,name = "count"))
df_word_dict = df_word_dict.sort_values(by = "count",ascending =False)

df_word_dict = df_word_dict[0:MAX_WORDS-2] #
df_word_dict["word_id"] = range(2,MAX_WORDS) #编号0和1分别留给未知词<unkown>和填充<padding>

word_id_dict = df_word_dict["word_id"].to_dict()

df_word_dict.head(10)
```

	count	word_id
the	268230	2
and	129713	3
a	129479	4
of	116497	5
to	108296	6
is	85615	7
	84074	8
in	74715	9
it	62587	10
i	60837	11

然后我们利用构建好的词典，将文本转换成token序号。

```
#转换token

# 填充文本
def pad(data_list,pad_length):
    padded_list = data_list.copy()
    if len(data_list)> pad_length:
         padded_list = data_list[-pad_length:]
    if len(data_list)< pad_length:
         padded_list = [1]*(pad_length-len(data_list))+data_list
    return padded_list

def text_to_token(text_file,token_file):
    with open(text_file,"r",encoding = 'utf-8') as fin,\
      open(token_file,"w",encoding = 'utf-8') as fout:
        for line in fin:
            label,text = line.split("\t")
            cleaned_text = clean_text(text)
            word_token_list = [word_id_dict.get(word, 0) for word in cleaned_text.split(" ")]
            pad_list = pad(word_token_list,MAX_LEN)
            out_line = label+"\t"+" ".join([str(x) for x in pad_list])
            fout.write(out_line+"\n")

text_to_token(train_data_path,train_token_path)
text_to_token(test_data_path,test_token_path)
```

接着将token文本按照样本分割，每个文件存放一个样本的数据。

```
# 分割样本
import os

if not os.path.exists(train_samples_path):
    os.mkdir(train_samples_path)

if not os.path.exists(test_samples_path):
    os.mkdir(test_samples_path)


def split_samples(token_path,samples_dir):
    with open(token_path,"r",encoding = 'utf-8') as fin:
        i = 0
        for line in fin:
            with open(samples_dir+"%d.txt"%i,"w",encoding = "utf-8") as fout:
                fout.write(line)
            i = i+1

split_samples(train_token_path,train_samples_path)
split_samples(test_token_path,test_samples_path)


print(os.listdir(train_samples_path)[0:100])
```

```
['11303.txt', '3644.txt', '19987.txt', '18441.txt', '5235.txt', '17772.txt', '1053.txt', '13514.txt', '8711.txt',
```

一切准备就绪，我们可以创建数据集Dataset, 从文件名称列表中读取文件内容了。

```
import os
class imdbDataset(Dataset):
    def __init__(self,samples_dir):
        self.samples_dir = samples_dir
        self.samples_paths = os.listdir(samples_dir)

    def __len__(self):
        return len(self.samples_paths)

    def __getitem__(self,index):
        path = self.samples_dir + self.samples_paths[index]
        with open(path,"r",encoding = "utf-8") as f:
            line = f.readline()
            label,tokens = line.split("\t")
            label = torch.tensor([float(label)],dtype = torch.float)
            feature = torch.tensor([int(x) for x in tokens.split(" ")],dtype = torch.long)
            return  (feature,label)
```

```
ds_train = imdbDataset(train_samples_path)
ds_test = imdbDataset(test_samples_path)


print(len(ds_train))
print(len(ds_test))
```

```
20000
5000
```

```
dl_train = DataLoader(ds_train,batch_size = BATCH_SIZE,shuffle = True,num_workers=4)
dl_test = DataLoader(ds_test,batch_size = BATCH_SIZE,num_workers=4)

for features,labels in dl_train:
    print(features)
    print(labels)
    break
```

```
tensor([[   1,    1,    1,  ...,   29,    8,    8],
        [  13,   11,  247,  ...,    0,    0,    8],
        [8587,  555,   12,  ...,    3,    0,    8],
        ...,
        [   1,    1,    1,  ...,    2,    0,    8],
        [ 618,   62,   25,  ...,   20,  204,    8],
        [   1,    1,    1,  ...,   71,   85,    8]])
tensor([[1.],
        [0.],
        [0.],
        [1.],
        [0.],
        [1.],
        [0.],
        [1.],
        [1.],
        [1.],
        [0.],
        [0.],
        [0.],
        [1.],
        [0.],
        [1.],
        [1.],
        [1.],
        [0.],
        [1.]])
```

最后构建模型测试一下数据集管道是否可用。

```
import torch
from torch import nn
import importlib
from torchkeras import Model,summary

class Net(Model):

    def __init__(self):
        super(Net, self).__init__()

        #设置padding_idx参数后将在训练过程中将填充的token始终赋值为0向量
        self.embedding = nn.Embedding(num_embeddings = MAX_WORDS,embedding_dim = 3,padding_idx =
        self.conv = nn.Sequential()
        self.conv.add_module("conv_1",nn.Conv1d(in_channels = 3,out_channels = 16,kernel_size =
        self.conv.add_module("pool_1",nn.MaxPool1d(kernel_size = 2))
        self.conv.add_module("relu_1",nn.ReLU())
        self.conv.add_module("conv_2",nn.Conv1d(in_channels = 16,out_channels = 128,kernel_size
        self.conv.add_module("pool_2",nn.MaxPool1d(kernel_size = 2))
        self.conv.add_module("relu_2",nn.ReLU())

        self.dense = nn.Sequential()
        self.dense.add_module("flatten",nn.Flatten())
        self.dense.add_module("linear",nn.Linear(6144,1))
        self.dense.add_module("sigmoid",nn.Sigmoid())

    def forward(self,x):
        x = self.embedding(x).transpose(1,2)
        x = self.conv(x)
        y = self.dense(x)
        return y

model = Net()
print(model)

model.summary(input_shape = (200,),input_dtype = torch.LongTensor)
```

```
Net(
  (embedding): Embedding(10000, 3, padding_idx=1)
  (conv): Sequential(
    (conv_1): Conv1d(3, 16, kernel_size=(5,), stride=(1,))
    (pool_1): MaxPool1d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (relu_1): ReLU()
    (conv_2): Conv1d(16, 128, kernel_size=(2,), stride=(1,))
    (pool_2): MaxPool1d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (relu_2): ReLU()
  )
  (dense): Sequential(
    (flatten): Flatten()
    (linear): Linear(in_features=6144, out_features=1, bias=True)
    (sigmoid): Sigmoid()
  )
)
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
         Embedding-1               [-1, 200, 3]          30,000
            Conv1d-2              [-1, 16, 196]             256
         MaxPool1d-3               [-1, 16, 98]               0
              ReLU-4               [-1, 16, 98]               0
            Conv1d-5              [-1, 128, 97]           4,224
         MaxPool1d-6              [-1, 128, 48]               0
              ReLU-7              [-1, 128, 48]               0
           Flatten-8                 [-1, 6144]               0
            Linear-9                    [-1, 1]           6,145
          Sigmoid-10                    [-1, 1]               0
================================================================
Total params: 40,625
Trainable params: 40,625
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.000763
Forward/backward pass size (MB): 0.287796
Params size (MB): 0.154972
Estimated Total Size (MB): 0.443531
----------------------------------------------------------------
```

```
# 编译模型
def accuracy(y_pred,y_true):
    y_pred = torch.where(y_pred>0.5,torch.ones_like(y_pred,dtype = torch.float32),
                         torch.zeros_like(y_pred,dtype = torch.float32))
    acc = torch.mean(1-torch.abs(y_true-y_pred))
    return acc

model.compile(loss_func = nn.BCELoss(),optimizer= torch.optim.Adagrad(model.parameters(),lr = 0.
             metrics_dict={"accuracy":accuracy})



# 训练模型
dfhistory = model.fit(10,dl_train,dl_val=dl_test,log_step_freq= 200)
```

```
Start Training ...

==========================================================================================2020-07-11 23:21:53
{'step': 200, 'loss': 0.956, 'accuracy': 0.521}
{'step': 400, 'loss': 0.823, 'accuracy': 0.53}
{'step': 600, 'loss': 0.774, 'accuracy': 0.545}
{'step': 800, 'loss': 0.747, 'accuracy': 0.56}
{'step': 1000, 'loss': 0.726, 'accuracy': 0.572}

 +-------+-------+----------+----------+--------------+
| epoch |  loss | accuracy | val_loss | val_accuracy |
+-------+-------+----------+----------+--------------+
|   1   | 0.726 |  0.572   |  0.661   |    0.613     |
+-------+-------+----------+----------+--------------+

==========================================================================================2020-07-11 23:22:20
{'step': 200, 'loss': 0.605, 'accuracy': 0.668}
{'step': 400, 'loss': 0.602, 'accuracy': 0.674}
{'step': 600, 'loss': 0.592, 'accuracy': 0.681}
{'step': 800, 'loss': 0.584, 'accuracy': 0.687}
{'step': 1000, 'loss': 0.575, 'accuracy': 0.696}

 +-------+-------+----------+----------+--------------+
| epoch |  loss | accuracy | val_loss | val_accuracy |
+-------+-------+----------+----------+--------------+
|   2   | 0.575 |  0.696   |  0.553   |    0.716     |
+-------+-------+----------+----------+--------------+

==========================================================================================2020-07-11 23:25:53
{'step': 200, 'loss': 0.294, 'accuracy': 0.877}
{'step': 400, 'loss': 0.299, 'accuracy': 0.875}
{'step': 600, 'loss': 0.298, 'accuracy': 0.875}
{'step': 800, 'loss': 0.296, 'accuracy': 0.876}
{'step': 1000, 'loss': 0.298, 'accuracy': 0.875}

 +-------+-------+----------+----------+--------------+
| epoch |  loss | accuracy | val_loss | val_accuracy |
+-------+-------+----------+----------+--------------+
|   10  | 0.298 |  0.875   |  0.464   |    0.795     |
+-------+-------+----------+----------+--------------+

==========================================================================================2020-07-11 23:26:19
Finished Training...
```

三，使用DataLoader加载数据集

DataLoader能够控制batch的大小，batch中元素的采样方法，以及将batch结果整理成模型所需输入形式的方法，并且能够使用多进程读取数据。

DataLoader的函数签名如下。

```
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,
)
```

一般情况下，我们仅仅会配置 dataset, batch_size, shuffle, num_workers, drop_last这五个参数，其他参数使用默认值即可。

DataLoader除了可以加载我们前面讲的 torch.utils.data.Dataset 外，还能够加载另外一种数据集 torch.utils.data.IterableDataset。

和Dataset数据集相当于一种列表结构不同，IterableDataset相当于一种迭代器结构。 它更加复杂，一般较少使用。

- dataset : 数据集
- batch_size: 批次大小
- shuffle: 是否乱序
- sampler: 样本采样函数，一般无需设置。
- batch_sampler: 批次采样函数，一般无需设置。
- num_workers: 使用多进程读取数据，设置的进程数。
- collate_fn: 整理一个批次数据的函数。
- pin_memory: 是否设置为锁业内存。默认为False，锁业内存不会使用虚拟内存(硬盘)，从锁业内存拷贝到GPU上速度会更快。
- drop_last: 是否丢弃最后一个样本数量不足batch_size批次数据。
- timeout: 加载一个数据批次的最长等待时间，一般无需设置。
- worker_init_fn: 每个worker中dataset的初始化函数，常用于 IterableDataset。一般不使用。

```
#构建输入数据管道
ds = TensorDataset(torch.arange(1,50))
dl = DataLoader(ds,
                batch_size = 10,
                shuffle= True,
                num_workers=2,
                drop_last = True)
#迭代数据
for batch, in dl:
    print(batch)
```

```
tensor([43, 44, 21, 36,  9,  5, 28, 16, 20, 14])
tensor([23, 49, 35, 38,  2, 34, 45, 18, 15, 40])
tensor([26,  6, 27, 39,  8,  4, 24, 19, 32, 17])
tensor([ 1, 29, 11, 47, 12, 22, 48, 42, 10,  7])
```

如果对本书内容理解上有需要进一步和作者交流的地方，欢迎在公众号"Python与算法之美"下留言。作者时间和精力有限，会酌情予以回复。

也可以在公众号后台回复关键字：**加群**，加入读者交流群和大家讨论。

5-2,模型层layers

深度学习模型一般由各种模型层组合而成。

torch.nn中内置了非常丰富的各种模型层。它们都属于nn.Module的子类，具备参数管理功能。

例如：

- nn.Linear, nn.Flatten, nn.Dropout, nn.BatchNorm2d
- nn.Conv2d,nn.AvgPool2d,nn.Conv1d,nn.ConvTranspose2d
- nn.Embedding,nn.GRU,nn.LSTM
- nn.Transformer

如果这些内置模型层不能够满足需求，我们也可以通过继承nn.Module基类构建自定义的模型层。

实际上，pytorch不区分模型和模型层，都是通过继承nn.Module进行构建。

因此，我们只要继承nn.Module基类并实现forward方法即可自定义模型层。

一，内置模型层

```
import numpy as np
import torch
from torch import nn
```

一些常用的内置模型层简单介绍如下。

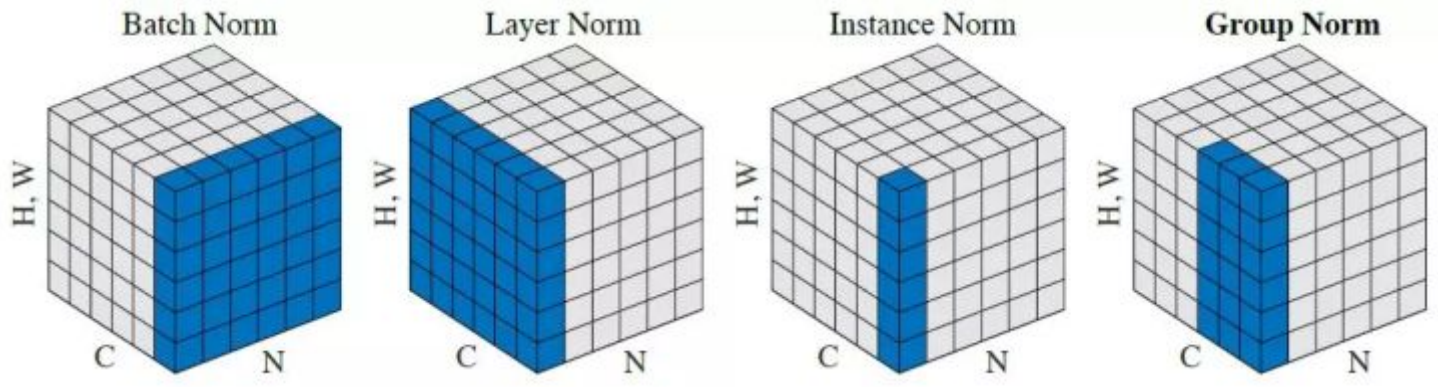
基础层

- nn.Linear：全连接层。参数个数 = 输入层特征数× 输出层特征数(weight) + 输出层特征数(bias)
- nn.Flatten：压平层，用于将多维张量样本压成一维张量样本。
- nn.BatchNorm1d：一维批标准化层。通过线性变换将输入批次缩放平移到稳定的均值和标准差。可以增强模型对输入不同分布的适应性，加快模型训练速度，有轻微正则化效果。一般在激活函数之前使用。可以用afine参数设置该层是否含有可以训练的参数。
- nn.BatchNorm2d：二维批标准化层。
- nn.BatchNorm3d：三维批标准化层。
- nn.Dropout：一维随机丢弃层。一种正则化手段。
- nn.Dropout2d：二维随机丢弃层。
- nn.Dropout3d：三维随机丢弃层。
- nn.Threshold：限幅层。当输入大于或小于阈值范围时，截断之。
- nn.ConstantPad2d： 二维常数填充层。对二维张量样本填充常数扩展长度。
- nn.ReplicationPad1d： 一维复制填充层。对一维张量样本通过复制边缘值填充扩展长度。
- nn.ZeroPad2d：二维零值填充层。对二维张量样本在边缘填充0值.
- nn.GroupNorm：组归一化。一种替代批归一化的方法，将通道分成若干组进行归一。不受batch大小限制，据称性能和效果都优于BatchNorm。
- nn.LayerNorm：层归一化。较少使用。

- nn.InstanceNorm2d: 样本归一化。较少使用。

各种归一化技术参考如下知乎文章《FAIR何恺明等人提出组归一化：替代批归一化，不受批量大小限制》

https://zhuanlan.zhihu.com/p/34858971



卷积网络相关层

- nn.Conv1d：普通一维卷积，常用于文本。参数个数 = 输入通道数×卷积核尺寸(如3)×卷积核个数 + 卷积核尺寸(如3)
- nn.Conv2d：普通二维卷积，常用于图像。参数个数 = 输入通道数×卷积核尺寸(如3乘3)×卷积核个数 + 卷积核尺寸(如3乘3)
  通过调整dilation参数大于1，可以变成空洞卷积，增大卷积核感受野。
  通过调整groups参数不为1，可以变成分组卷积。分组卷积中不同分组使用相同的卷积核，显著减少参数数量。
  当groups参数等于通道数时，相当于tensorflow中的二维深度卷积层 tf.keras.layers.DepthwiseConv2D。
  利用分组卷积和1乘1卷积的组合操作，可以构造相当于Keras中的二维深度可分离卷积层 tf.keras.layers.SeparableConv2D。
- nn.Conv3d：普通三维卷积，常用于视频。参数个数 = 输入通道数×卷积核尺寸(如3乘3乘3)×卷积核个数 + 卷积核尺寸(如3乘3乘3) 。
- nn.MaxPool1d: 一维最大池化。
- nn.MaxPool2d：二维最大池化。一种下采样方式。没有需要训练的参数。
- nn.MaxPool3d：三维最大池化。
- nn.AdaptiveMaxPool2d：二维自适应最大池化。无论输入图像的尺寸如何变化，输出的图像尺寸是固定的。
  该函数的实现原理，大概是通过输入图像的尺寸和要得到的输出图像的尺寸来反向推算池化算子的padding,stride等参数。
- nn.FractionalMaxPool2d：二维分数最大池化。普通最大池化通常输入尺寸是输出的整数倍。而分数最大池化则可以不必是整数。分数最大池化使用了一些随机采样策略，有一定的正则效果，可以用它来代替普通最大池化和Dropout层。

- nn.AvgPool2d：二维平均池化。
- nn.AdaptiveAvgPool2d：二维自适应平均池化。无论输入的维度如何变化，输出的维度是固定的。
- nn.ConvTranspose2d：二维卷积转置层，俗称反卷积层。并非卷积的逆操作，但在卷积核相同的情况下，当其输入尺寸是卷积操作输出尺寸的情况下，卷积转置的输出尺寸恰好是卷积操作的输入尺寸。在语义分割中可用于上采样。
- nn.Upsample：上采样层，操作效果和池化相反。可以通过mode参数控制上采样策略为"nearest"最邻近策略或"linear"线性插值策略。
- nn.Unfold：滑动窗口提取层。其参数和卷积操作nn.Conv2d相同。实际上，卷积操作可以等价于nn.Unfold和nn.Linear以及nn.Fold的一个组合。
  其中nn.Unfold操作可以从输入中提取各个滑动窗口的数值矩阵，并将其压平成一维。利用nn.Linear将nn.Unfold的输出和卷积核做乘法后，再使用
  nn.Fold操作将结果转换成输出图片形状。
- nn.Fold：逆滑动窗口提取层。

循环网络相关层

- nn.Embedding：嵌入层。一种比Onehot更加有效的对离散特征进行编码的方法。一般用于将输入中的单词映射为稠密向量。嵌入层的参数需要学习。
- nn.LSTM：长短记忆循环网络层【支持多层】。最普遍使用的循环网络层。具有携带轨道，遗忘门，更新门，输出门。可以较为有效地缓解梯度消失问题，从而能够适用长期依赖问题。设置bidirectional = True时可以得到双向LSTM。需要注意的时，默认的输入和输出形状是(seq,batch,feature), 如果需要将batch维度放在第0维，则要设置batch_first参数设置为True。
- nn.GRU：门控循环网络层【支持多层】。LSTM的低配版，不具有携带轨道，参数数量少于LSTM，训练速度更快。
- nn.RNN：简单循环网络层【支持多层】。容易存在梯度消失，不能够适用长期依赖问题。一般较少使用。
- nn.LSTMCell：长短记忆循环网络单元。和nn.LSTM在整个序列上迭代相比，它仅在序列上迭代一步。一般较少使用。
- nn.GRUCell：门控循环网络单元。和nn.GRU在整个序列上迭代相比，它仅在序列上迭代一步。一般较少使用。
- nn.RNNCell：简单循环网络单元。和nn.RNN在整个序列上迭代相比，它仅在序列上迭代一步。一般较少使用。

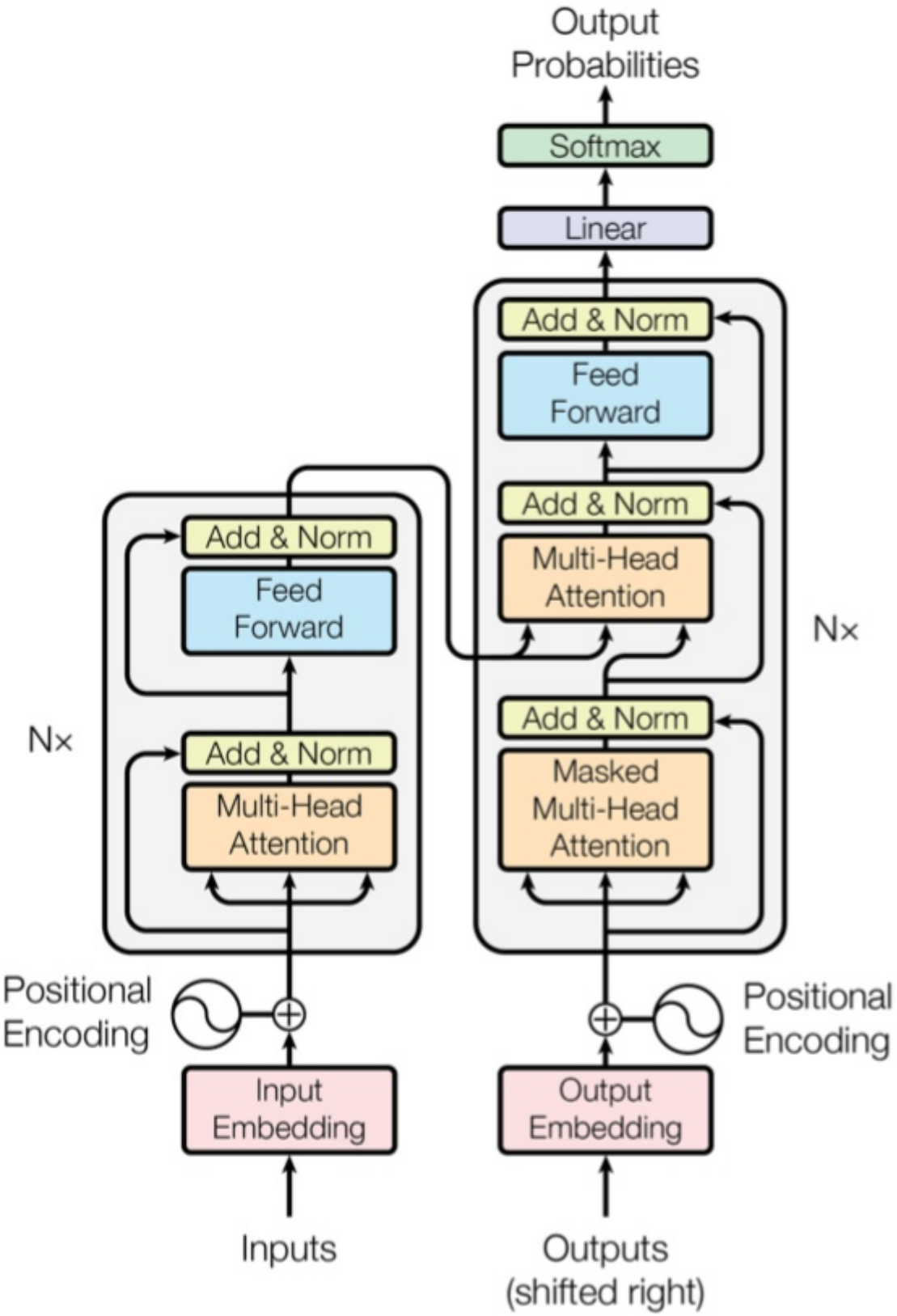
Transformer相关层

- nn.Transformer：Transformer网络结构。Transformer网络结构是替代循环网络的一种结构，解决了循环网络难以并行，难以捕捉长期依赖的缺陷。它是目前NLP任务的主流模型的主要构成部分。Transformer网络结构由TransformerEncoder编码器和TransformerDecoder解码器组成。编码器和解码器的核心是MultiheadAttention多头注意力层。

- nn.TransformerEncoder：Transformer编码器结构。由多个 nn.TransformerEncoderLayer编码器层组成。
- nn.TransformerDecoder：Transformer解码器结构。由多个 nn.TransformerDecoderLayer解码器层组成。
- nn.TransformerEncoderLayer：Transformer的编码器层。
- nn.TransformerDecoderLayer：Transformer的解码器层。
- nn.MultiheadAttention：多头注意力层。

Transformer原理介绍可以参考如下知乎文章《详解Transformer(Attention Is All You Need)》

https://zhuanlan.zhihu.com/p/48508221



二，自定义模型层

如果Pytorch的内置模型层不能够满足需求，我们也可以通过继承nn.Module基类构建自定义的模型层。

实际上，pytorch不区分模型和模型层，都是通过继承nn.Module进行构建。

因此，我们只要继承nn.Module基类并实现forward方法即可自定义模型层。

下面是Pytorch的nn.Linear层的源码，我们可以仿照它来自定义模型层。

```
import torch
from torch import nn
import torch.nn.functional as F


class Linear(nn.Module):
    __constants__ = ['in_features', 'out_features']

    def __init__(self, in_features, out_features, bias=True):
        super(Linear, self).__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.weight = nn.Parameter(torch.Tensor(out_features, in_features))
        if bias:
            self.bias = nn.Parameter(torch.Tensor(out_features))
        else:
            self.register_parameter('bias', None)
        self.reset_parameters()

    def reset_parameters(self):
        nn.init.kaiming_uniform_(self.weight, a=math.sqrt(5))
        if self.bias is not None:
            fan_in, _ = nn.init._calculate_fan_in_and_fan_out(self.weight)
            bound = 1 / math.sqrt(fan_in)
            nn.init.uniform_(self.bias, -bound, bound)

    def forward(self, input):
        return F.linear(input, self.weight, self.bias)

    def extra_repr(self):
        return 'in_features={}, out_features={}, bias={}'.format(
            self.in_features, self.out_features, self.bias is not None
        )


linear = nn.Linear(20, 30)
inputs = torch.randn(128, 20)
output = linear(inputs)
print(output.size())
```

```
torch.Size([128, 30])
```

如果对本书内容理解上有需要进一步和作者交流的地方，欢迎在公众号"Python与算法之美"下留言。作者时间和精力有限，会酌情予以回复。

也可以在公众号后台回复关键字：**加群**，加入读者交流群和大家讨论。

5-5,损失函数losses

一般来说，监督学习的目标函数由损失函数和正则化项组成。(Objective = Loss + Regularization)

Pytorch中的损失函数一般在训练模型时候指定。

注意Pytorch中内置的损失函数的参数和tensorflow不同，是y_pred在前，y_true在后，而Tensorflow是y_true在前，y_pred在后。

对于回归模型，通常使用的内置损失函数是均方损失函数nn.MSELoss 。

对于二分类模型，通常使用的是二元交叉熵损失函数nn.BCELoss (输入已经是sigmoid激活函数之后的结果)
或者 nn.BCEWithLogitsLoss (输入尚未经过nn.Sigmoid激活函数) 。

对于多分类模型，一般推荐使用交叉熵损失函数 nn.CrossEntropyLoss。
(y_true需要是一维的，是类别编码。y_pred未经过nn.Softmax激活。)

此外，如果多分类的y_pred经过了nn.LogSoftmax激活，可以使用nn.NLLLoss损失函数(The negative log likelihood loss)。
这种方法和直接使用nn.CrossEntropyLoss等价。

如果有需要，也可以自定义损失函数，自定义损失函数需要接收两个张量y_pred，y_true作为输入参数，并输出一个标量作为损失函数值。

Pytorch中的正则化项一般通过自定义的方式和损失函数一起添加作为目标函数。

如果仅仅使用L2正则化，也可以利用优化器的weight_decay参数来实现相同的效果。

一，内置损失函数

```
import numpy as np
import pandas as pd
import torch
from torch import nn
import torch.nn.functional as F


y_pred = torch.tensor([[10.0,0.0,-10.0],[8.0,8.0,8.0]])
y_true = torch.tensor([0,2])

# 直接调用交叉熵损失
ce = nn.CrossEntropyLoss()(y_pred,y_true)
print(ce)

# 等价于先计算nn.LogSoftmax激活，再调用NLLLoss
y_pred_logsoftmax = nn.LogSoftmax(dim = 1)(y_pred)
nll = nn.NLLLoss()(y_pred_logsoftmax,y_true)
print(nll)
```

```
tensor(0.5493)
tensor(0.5493)
```

内置的损失函数一般有类的实现和函数的实现两种形式。

如：nn.BCE 和 F.binary_cross_entropy 都是二元交叉熵损失函数，前者是类的实现形式，后者是函数的实现形式。

实际上类的实现形式通常是调用函数的实现形式并用nn.Module封装后得到的。

一般我们常用的是类的实现形式。它们封装在torch.nn模块下，并且类名以Loss结尾。

常用的一些内置损失函数说明如下。

- nn.MSELoss（均方误差损失，也叫做L2损失，用于回归）
- nn.L1Loss （L1损失，也叫做绝对值误差损失，用于回归）
- nn.SmoothL1Loss (平滑L1损失，当输入在-1到1之间时，平滑为L2损失，用于回归)

- nn.BCELoss (二元交叉熵，用于二分类，输入已经过nn.Sigmoid激活，对不平衡数据集可以用weigths参数调整类别权重)
- nn.BCEWithLogitsLoss (二元交叉熵，用于二分类，输入未经过nn.Sigmoid激活)
- nn.CrossEntropyLoss (交叉熵，用于多分类，要求label为稀疏编码，输入未经过nn.Softmax激活，对不平衡数据集可以用weigths参数调整类别权重)
- nn.NLLLoss (负对数似然损失，用于多分类，要求label为稀疏编码，输入经过nn.LogSoftmax激活)
- nn.CosineSimilarity(余弦相似度，可用于多分类)
- nn.AdaptiveLogSoftmaxWithLoss (一种适合非常多类别且类别分布很不均衡的损失函数，会自适应地将多个小类别合成一个cluster)

更多损失函数的介绍参考如下知乎文章：

《PyTorch的十八个损失函数》

https://zhuanlan.zhihu.com/p/61379965

二，自定义损失函数

自定义损失函数接收两个张量y_pred,y_true作为输入参数，并输出一个标量作为损失函数值。

也可以对nn.Module进行子类化，重写forward方法实现损失的计算逻辑，从而得到损失函数的类的实现。

下面是一个Focal Loss的自定义实现示范。Focal Loss是一种对binary_crossentropy的改进损失函数形式。

它在样本不均衡和存在较多易分类的样本时相比binary_crossentropy具有明显的优势。

它有两个可调参数，alpha参数和gamma参数。其中alpha参数主要用于衰减负样本的权重，gamma参数主要用于衰减容易训练样本的权重。

从而让模型更加聚焦在正样本和困难样本上。这就是为什么这个损失函数叫做Focal Loss。

详见《5分钟理解Focal Loss与GHM——解决样本不平衡利器》

https://zhuanlan.zhihu.com/p/80594704

$$focal\_loss(y,p) = \begin{cases} -\alpha(1-p)^{\gamma}\log(p) & \text{if y} = 1 \\ -(1-\alpha)p^{\gamma}\log(1-p) & \text{if y} = 0 \end{cases}$$

```python
class FocalLoss(nn.Module):

    def __init__(self,gamma=2.0,alpha=0.75):
        super().__init__()
        self.gamma = gamma
        self.alpha = alpha

    def forward(self,y_pred,y_true):
        bce = torch.nn.BCELoss(reduction = "none")(y_pred,y_true)
        p_t = (y_true * y_pred) + ((1 - y_true) * (1 - y_pred))
        alpha_factor = y_true * self.alpha + (1 - y_true) * (1 - self.alpha)
        modulating_factor = torch.pow(1.0 - p_t, self.gamma)
        loss = torch.mean(alpha_factor * modulating_factor * bce)
        return loss
```

```python
#困难样本
y_pred_hard = torch.tensor([[0.5],[0.5]])
y_true_hard = torch.tensor([[1.0],[0.0]])

#容易样本
y_pred_easy = torch.tensor([[0.9],[0.1]])
y_true_easy = torch.tensor([[1.0],[0.0]])

focal_loss = FocalLoss()
bce_loss = nn.BCELoss()

print("focal_loss(hard samples):", focal_loss(y_pred_hard,y_true_hard))
print("bce_loss(hard samples):", bce_loss(y_pred_hard,y_true_hard))
print("focal_loss(easy samples):", focal_loss(y_pred_easy,y_true_easy))
print("bce_loss(easy samples):", bce_loss(y_pred_easy,y_true_easy))

#可见 focal_loss让容易样本的权重衰减到原来的 0.0005/0.1054 = 0.00474
#而让困难样本的权重只衰减到原来的 0.0866/0.6931=0.12496

# 因此相对而言，focal_loss可以衰减容易样本的权重。
```

```
focal_loss(hard samples): tensor(0.0866)
bce_loss(hard samples): tensor(0.6931)
focal_loss(easy samples): tensor(0.0005)
bce_loss(easy samples): tensor(0.1054)
```

FocalLoss的使用完整范例可以参考下面中 自定义L1和L2正则化项 中的范例，该范例既演示了自定义正则化项的方法，也演示了FocalLoss的使用方法。

三，自定义L1和L2正则化项

通常认为L1 正则化可以产生稀疏权值矩阵，即产生一个稀疏模型，可以用于特征选择。

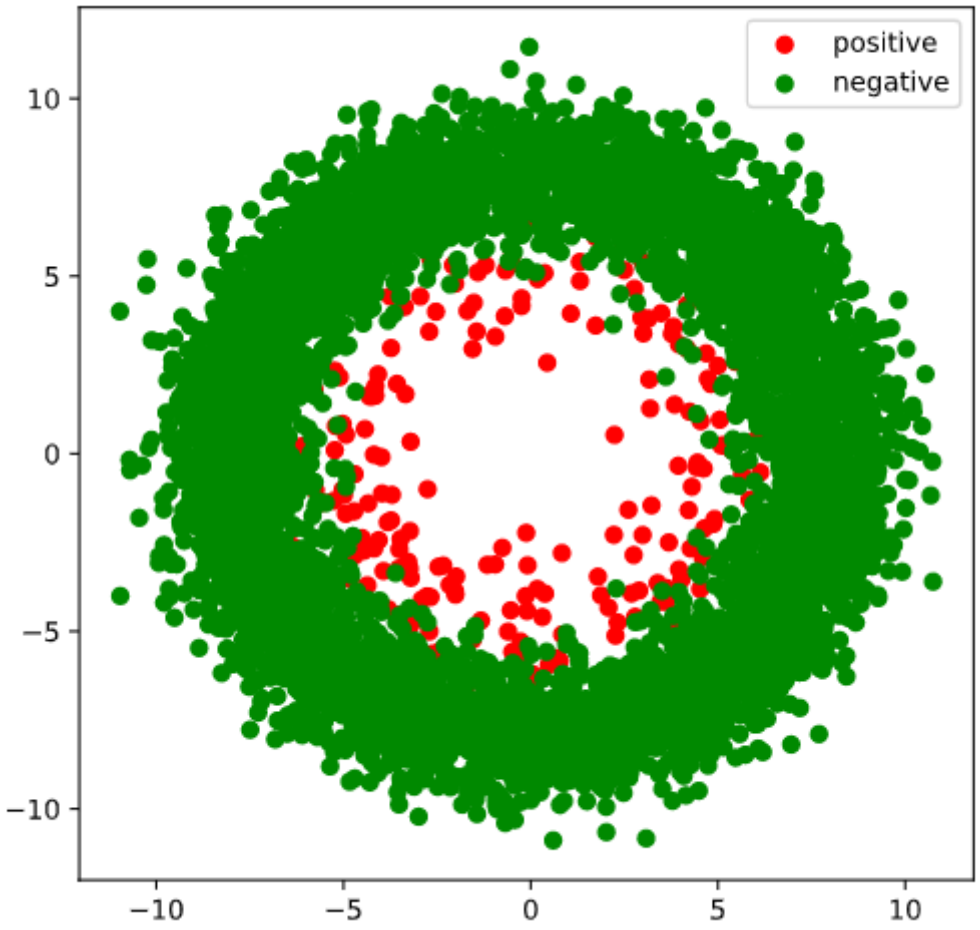
而L2 正则化可以防止模型过拟合（overfitting）。一定程度上，L1也可以防止过拟合。

下面以一个二分类问题为例，演示给模型的目标函数添加自定义L1和L2正则化项的方法。

这个范例同时演示了上一个部分的FocalLoss的使用。

1，准备数据

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
import torch
from torch import nn
import torch.nn.functional as F
from torch.utils.data import Dataset,DataLoader,TensorDataset
import torchkeras
%matplotlib inline
%config InlineBackend.figure_format = 'svg'

#正负样本数量
n_positive,n_negative = 200,6000

#生成正样本，小圆环分布
r_p = 5.0 + torch.normal(0.0,1.0,size = [n_positive,1])
theta_p = 2*np.pi*torch.rand([n_positive,1])
Xp = torch.cat([r_p*torch.cos(theta_p),r_p*torch.sin(theta_p)],axis = 1)
Yp = torch.ones_like(r_p)

#生成负样本，大圆环分布
r_n = 8.0 + torch.normal(0.0,1.0,size = [n_negative,1])
theta_n = 2*np.pi*torch.rand([n_negative,1])
Xn = torch.cat([r_n*torch.cos(theta_n),r_n*torch.sin(theta_n)],axis = 1)
Yn = torch.zeros_like(r_n)

#汇总样本
X = torch.cat([Xp,Xn],axis = 0)
Y = torch.cat([Yp,Yn],axis = 0)


#可视化
plt.figure(figsize = (6,6))
plt.scatter(Xp[:,0],Xp[:,1],c = "r")
plt.scatter(Xn[:,0],Xn[:,1],c = "g")
plt.legend(["positive","negative"]);
```




```
ds = TensorDataset(X,Y)

ds_train,ds_valid = torch.utils.data.random_split(ds,[int(len(ds)*0.7),len(ds)-int(len(ds)*0.7)])
dl_train = DataLoader(ds_train,batch_size = 100,shuffle=True,num_workers=2)
dl_valid = DataLoader(ds_valid,batch_size = 100,num_workers=2)
```

2，定义模型

```
class DNNModel(torchkeras.Model):
    def __init__(self):
        super(DNNModel, self).__init__()
        self.fc1 = nn.Linear(2,4)
        self.fc2 = nn.Linear(4,8)
        self.fc3 = nn.Linear(8,1)

    def forward(self,x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        y = nn.Sigmoid()(self.fc3(x))
        return y

model = DNNModel()

model.summary(input_shape =(2,))
```

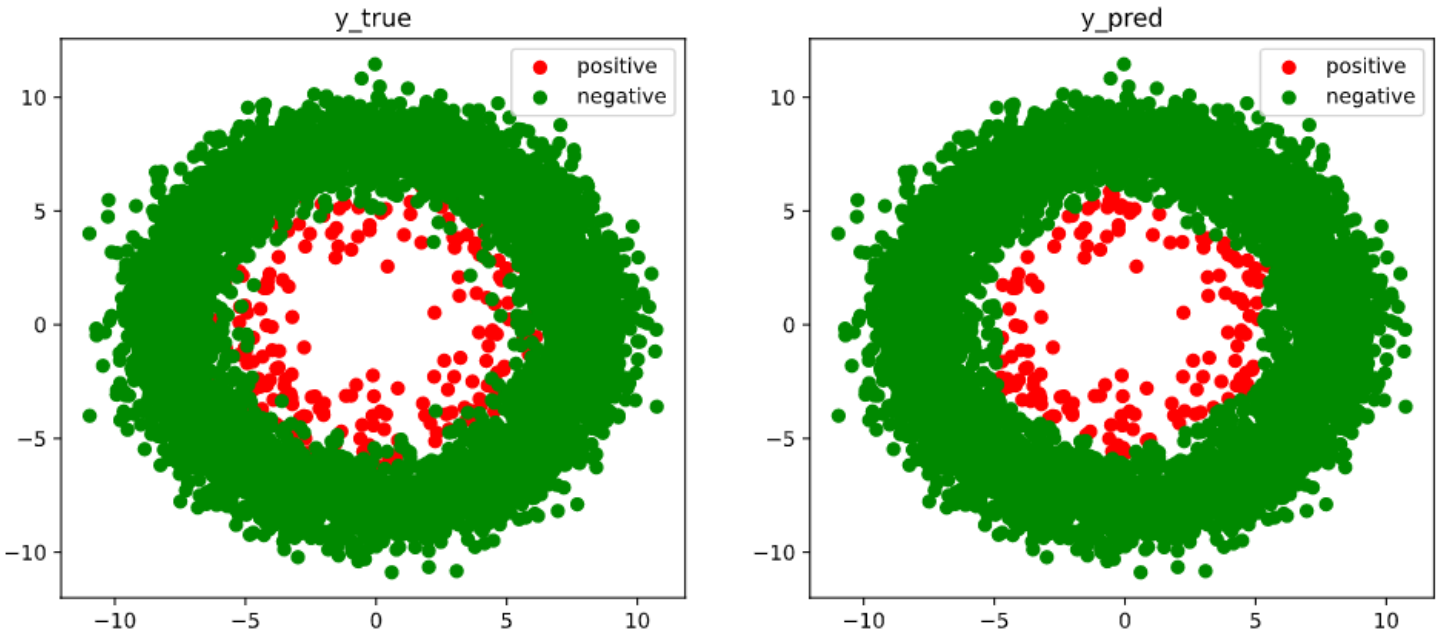
```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1                    [-1, 4]              12
            Linear-2                    [-1, 8]              40
            Linear-3                    [-1, 1]               9
================================================================
Total params: 61
Trainable params: 61
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.000008
Forward/backward pass size (MB): 0.000099
Params size (MB): 0.000233
Estimated Total Size (MB): 0.000340
----------------------------------------------------------------
```

3，训练模型

```
# 准确率
def accuracy(y_pred,y_true):
    y_pred = torch.where(y_pred>0.5,torch.ones_like(y_pred,dtype = torch.float32),
                         torch.zeros_like(y_pred,dtype = torch.float32))
    acc = torch.mean(1-torch.abs(y_true-y_pred))
    return acc

# L2正则化
def L2Loss(model,alpha):
    l2_loss = torch.tensor(0.0, requires_grad=True)
    for name, param in model.named_parameters():
        if 'bias' not in name: #一般不对偏置项使用正则
            l2_loss = l2_loss + (0.5 * alpha * torch.sum(torch.pow(param, 2)))
    return l2_loss

# L1正则化
def L1Loss(model,beta):
    l1_loss = torch.tensor(0.0, requires_grad=True)
    for name, param in model.named_parameters():
        if 'bias' not in name:
            l1_loss = l1_loss +  beta * torch.sum(torch.abs(param))
    return l1_loss

# 将L2正则和L1正则添加到FocalLoss损失，一起作为目标函数
def focal_loss_with_regularization(y_pred,y_true):
    focal = FocalLoss()(y_pred,y_true)
    l2_loss = L2Loss(model,0.001) #注意设置正则化项系数
    l1_loss = L1Loss(model,0.001)
    total_loss = focal + l2_loss + l1_loss
    return total_loss

model.compile(loss_func =focal_loss_with_regularization,
              optimizer= torch.optim.Adam(model.parameters(),lr = 0.01),
             metrics_dict={"accuracy":accuracy})

dfhistory = model.fit(30,dl_train = dl_train,dl_val = dl_valid,log_step_freq = 30)
```

```
Start Training ...

==========================================================================================2020-07-11 23:34:17
{'step': 30, 'loss': 0.021, 'accuracy': 0.972}

 +-------+-------+----------+----------+--------------+
| epoch |  loss | accuracy | val_loss | val_accuracy |
+-------+-------+----------+----------+--------------+
|   1   | 0.022 |  0.971   |  0.025   |     0.96     |
+-------+-------+----------+----------+--------------+

==========================================================================================2020-07-11 23:34:27
{'step': 30, 'loss': 0.016, 'accuracy': 0.984}

 +-------+-------+----------+----------+--------------+
| epoch |  loss | accuracy | val_loss | val_accuracy |
+-------+-------+----------+----------+--------------+
|   30  | 0.016 |  0.981   |  0.017   |    0.983     |
+-------+-------+----------+----------+--------------+

==========================================================================================2020-07-11 23:34:27
Finished Training...
```

```
# 结果可视化
fig, (ax1,ax2) = plt.subplots(nrows=1,ncols=2,figsize = (12,5))
ax1.scatter(Xp[:,0],Xp[:,1], c="r")
ax1.scatter(Xn[:,0],Xn[:,1],c = "g")
ax1.legend(["positive","negative"]);
ax1.set_title("y_true");

Xp_pred = X[torch.squeeze(model.forward(X)>=0.5)]
Xn_pred = X[torch.squeeze(model.forward(X)<0.5)]

ax2.scatter(Xp_pred[:,0],Xp_pred[:,1],c = "r")
ax2.scatter(Xn_pred[:,0],Xn_pred[:,1],c = "g")
ax2.legend(["positive","negative"]);
ax2.set_title("y_pred");
```



四，通过优化器实现L2正则化

如果仅仅需要使用L2正则化，那么也可以利用优化器的weight_decay参数来实现。

weight_decay参数可以设置参数在训练过程中的衰减，这和L2正则化的作用效果等价。

```
before L2 regularization:

gradient descent: w = w - lr * dloss_dw

after L2 regularization:

gradient descent: w = w - lr * (dloss_dw+beta*w) = (1-lr*beta)*w - lr*dloss_dw

so  (1-lr*beta) is the weight decay ratio.
```

Pytorch的优化器支持一种称之为Per-parameter options的操作，就是对每一个参数进行特定的学习率，权重衰减率指定，以满足更为细致的要求。

```
weight_params = [param for name, param in model.named_parameters() if "bias" not in name]
bias_params = [param for name, param in model.named_parameters() if "bias" in name]

optimizer = torch.optim.SGD([{'params': weight_params, 'weight_decay':1e-5},
                             {'params': bias_params, 'weight_decay':0}],
                            lr=1e-2, momentum=0.9)
```

如果对本书内容理解上有需要进一步和作者交流的地方，欢迎在公众号"Python与算法之美"下留言。作者时间和精力有限，会酌情予以回复。

也可以在公众号后台回复关键字：**加群**，加入读者交流群和大家讨论。

5-4,TensorBoard可视化

在我们的炼丹过程中，如果能够使用丰富的图像来展示模型的结构，指标的变化，参数的分布，输入的形态等信息，无疑会提升我们对问题的洞察力，并增加许多炼丹的乐趣。

TensorBoard正是这样一个神奇的炼丹可视化辅助工具。它原是TensorFlow的小弟，但它也能够很好地和Pytorch进行配合。甚至在Pytorch中使用TensorBoard比TensorFlow中使用TensorBoard还要来的更加简单和自然。

Pytorch中利用TensorBoard可视化的大概过程如下：

首先在Pytorch中指定一个目录创建一个torch.utils.tensorboard.SummaryWriter日志写入器。

然后根据需要可视化的信息，利用日志写入器将相应信息日志写入我们指定的目录。

最后就可以传入日志目录作为参数启动TensorBoard，然后就可以在TensorBoard中愉快地看片了。

我们主要介绍Pytorch中利用TensorBoard进行如下方面信息的可视化的方法。

- 可视化模型结构： writer.add_graph
- 可视化指标变化： writer.add_scalar
- 可视化参数分布： writer.add_histogram

- 可视化原始图像： writer.add_image 或 writer.add_images
- 可视化人工绘图： writer.add_figure

一，可视化模型结构

```
import torch
from torch import nn
from torch.utils.tensorboard import SummaryWriter
from torchkeras import Model,summary


class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=3,out_channels=32,kernel_size = 3)
        self.pool = nn.MaxPool2d(kernel_size = 2,stride = 2)
        self.conv2 = nn.Conv2d(in_channels=32,out_channels=64,kernel_size = 5)
        self.dropout = nn.Dropout2d(p = 0.1)
        self.adaptive_pool = nn.AdaptiveMaxPool2d((1,1))
        self.flatten = nn.Flatten()
        self.linear1 = nn.Linear(64,32)
        self.relu = nn.ReLU()
        self.linear2 = nn.Linear(32,1)
        self.sigmoid = nn.Sigmoid()

    def forward(self,x):
        x = self.conv1(x)
        x = self.pool(x)
        x = self.conv2(x)
        x = self.pool(x)
        x = self.dropout(x)
        x = self.adaptive_pool(x)
        x = self.flatten(x)
        x = self.linear1(x)
        x = self.relu(x)
        x = self.linear2(x)
        y = self.sigmoid(x)
        return y

net = Net()
print(net)
```

```
Net(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1))
  (dropout): Dropout2d(p=0.1, inplace=False)
  (adaptive_pool): AdaptiveMaxPool2d(output_size=(1, 1))
  (flatten): Flatten()
  (linear1): Linear(in_features=64, out_features=32, bias=True)
  (relu): ReLU()
  (linear2): Linear(in_features=32, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)
```

```
summary(net,input_shape= (3,32,32))
```

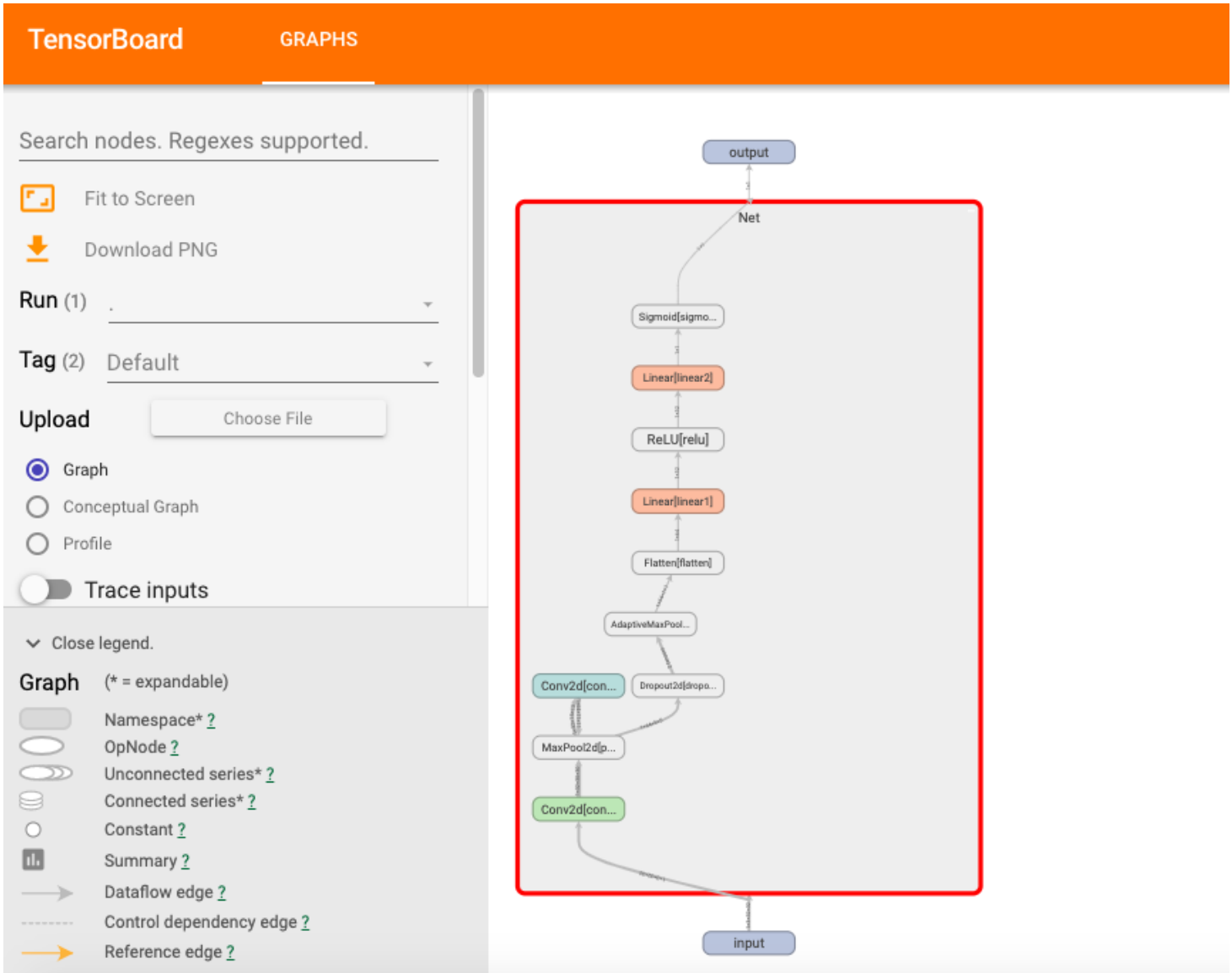
```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 30, 30]             896
         MaxPool2d-2           [-1, 32, 15, 15]               0
            Conv2d-3           [-1, 64, 11, 11]          51,264
         MaxPool2d-4             [-1, 64, 5, 5]               0
         Dropout2d-5             [-1, 64, 5, 5]               0
 AdaptiveMaxPool2d-6             [-1, 64, 1, 1]               0
           Flatten-7                   [-1, 64]               0
            Linear-8                   [-1, 32]           2,080
              ReLU-9                   [-1, 32]               0
           Linear-10                    [-1, 1]              33
          Sigmoid-11                    [-1, 1]               0
================================================================
Total params: 54,273
Trainable params: 54,273
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.011719
Forward/backward pass size (MB): 0.359634
Params size (MB): 0.207035
Estimated Total Size (MB): 0.578388
----------------------------------------------------------------
```

```
writer = SummaryWriter('./data/tensorboard')
writer.add_graph(net,input_to_model = torch.rand(1,3,32,32))
writer.close()
```

```
%load_ext tensorboard
#%tensorboard --logdir ./data/tensorboard
```

```
from tensorboard import notebook
#查看启动的tensorboard程序
notebook.list()
```

```
#启动tensorboard程序
notebook.start("--logdir ./data/tensorboard")
#等价于在命令行中执行 tensorboard --logdir ./data/tensorboard
#可以在浏览器中打开 http://localhost:6006/ 查看
```



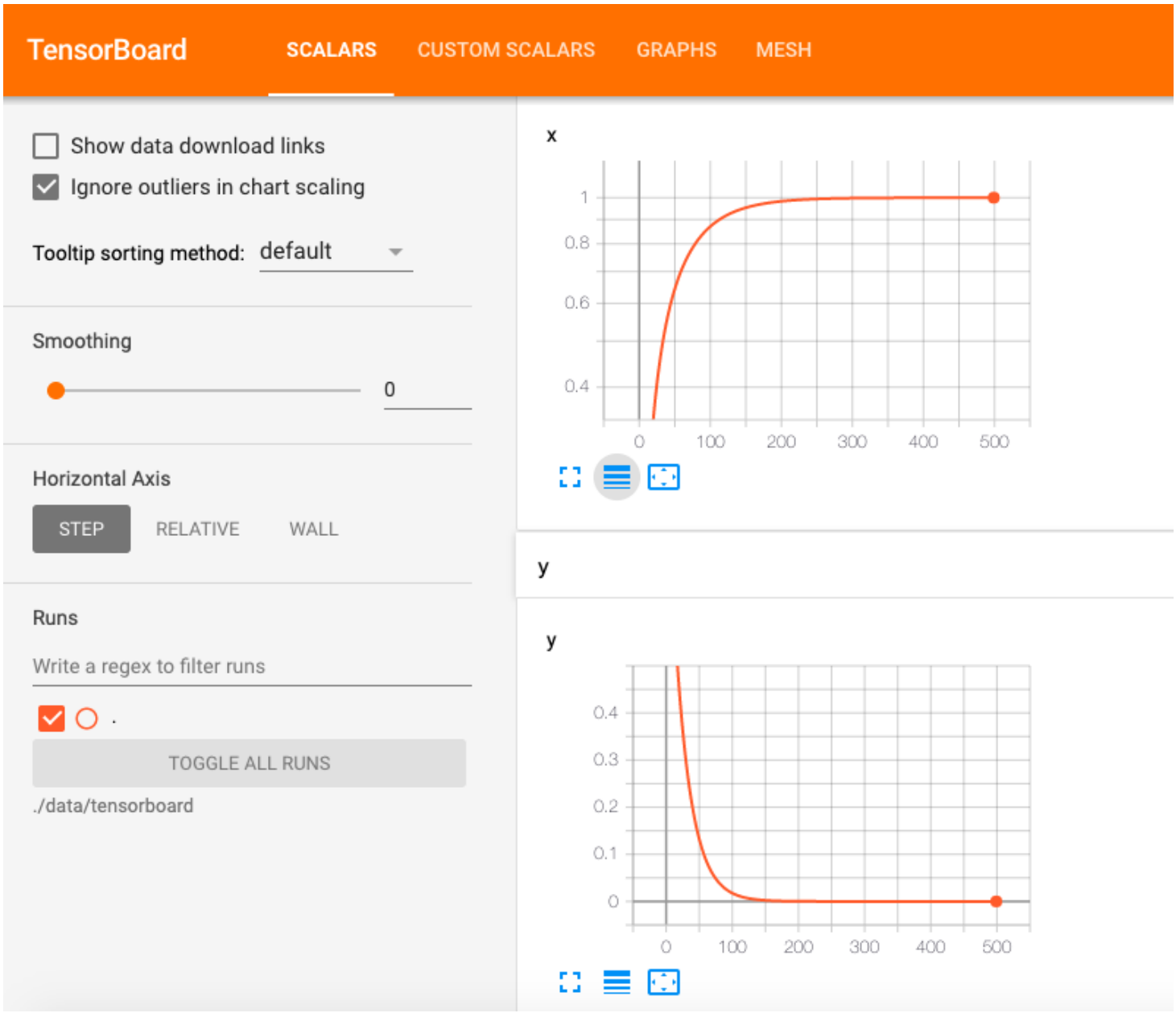
二，可视化指标变化

有时候在训练过程中，如果能够实时动态地查看loss和各种metric的变化曲线，那么无疑可以帮助我们更加直观地了解模型的训练情况。

注意，writer.add_scalar仅能对标量的值的变化进行可视化。因此它一般用于对loss和metric的变化进行可视化分析。

```
import numpy as np
import torch
from torch.utils.tensorboard import SummaryWriter



# 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)


def f(x):
    result = a*torch.pow(x,2) + b*x + c
    return(result)

writer = SummaryWriter('./data/tensorboard')
for i in range(500):
    optimizer.zero_grad()
    y = f(x)
    y.backward()
    optimizer.step()
    writer.add_scalar("x",x.item(),i) #日志中记录x在第step i 的值
    writer.add_scalar("y",y.item(),i) #日志中记录y在第step i 的值

writer.close()

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

```
y= tensor(0.) ; x= tensor(1.0000)
```

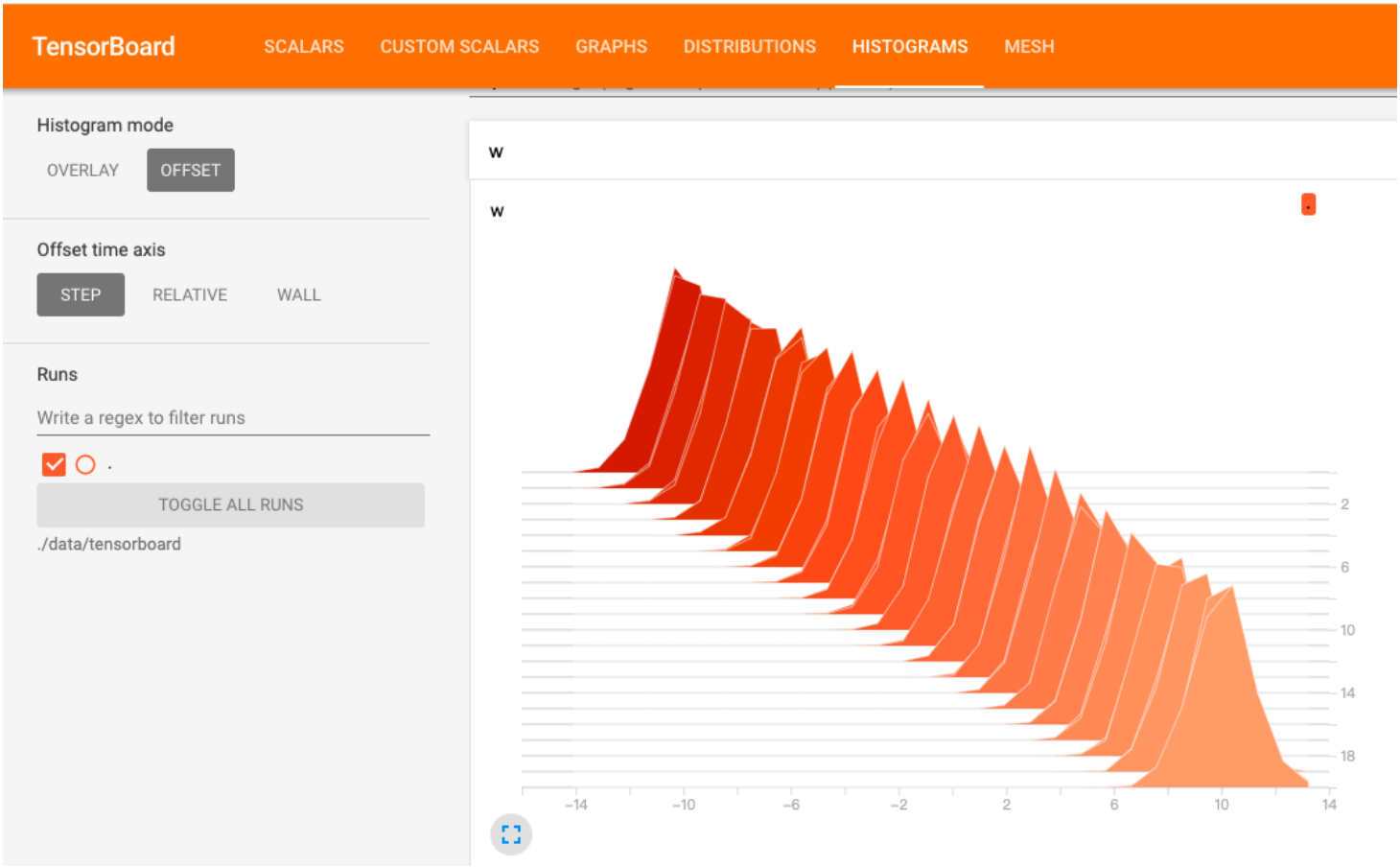


三，可视化参数分布

如果需要对模型的参数(一般非标量)在训练过程中的变化进行可视化，可以使用 writer.add_histogram。

它能够观测张量值分布的直方图随训练步骤的变化趋势。

```
import numpy as np
import torch
from torch.utils.tensorboard import SummaryWriter


# 创建正态分布的张量模拟参数矩阵
def norm(mean,std):
    t = std*torch.randn((100,20))+mean
    return t

writer = SummaryWriter('./data/tensorboard')
for step,mean in enumerate(range(-10,10,1)):
    w = norm(mean,1)
    writer.add_histogram("w",w, step)
    writer.flush()
writer.close()
```



四，可视化原始图像

如果我们做图像相关的任务，也可以将原始的图片在tensorboard中进行可视化展示。

如果只写入一张图片信息，可以使用writer.add_image。

如果要写入多张图片信息，可以使用writer.add_images。

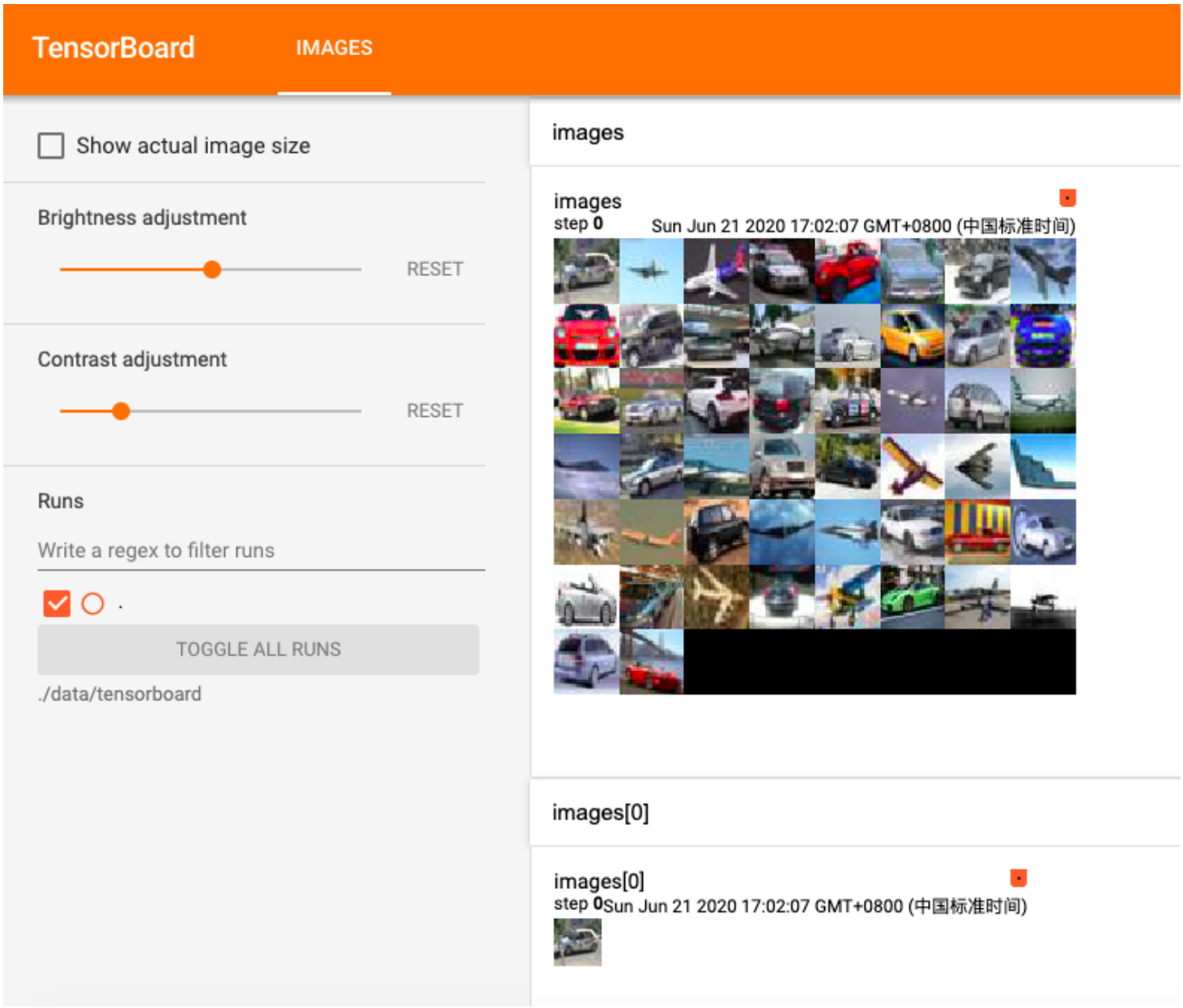
也可以用 torchvision.utils.make_grid将多张图片拼成一张图片，然后用writer.add_image写入。

注意，传入的是代表图片信息的Pytorch中的张量数据。

```
import torch
import torchvision
from torch import nn
from torch.utils.data import Dataset,DataLoader
from torchvision import transforms,datasets


transform_train = transforms.Compose(
    [transforms.ToTensor()])
transform_valid = transforms.Compose(
    [transforms.ToTensor()])
```

```
ds_train = datasets.ImageFolder("./data/cifar2/train/",
            transform = transform_train,target_transform= lambda t:torch.tensor([t]).float())
ds_valid = datasets.ImageFolder("./data/cifar2/test/",
            transform = transform_train,target_transform= lambda t:torch.tensor([t]).float())

print(ds_train.class_to_idx)

dl_train = DataLoader(ds_train,batch_size = 50,shuffle = True,num_workers=3)
dl_valid = DataLoader(ds_valid,batch_size = 50,shuffle = True,num_workers=3)

dl_train_iter = iter(dl_train)
images, labels = dl_train_iter.next()

# 仅查看一张图片
writer = SummaryWriter('./data/tensorboard')
writer.add_image('images[0]', images[0])
writer.close()

# 将多张图片拼接成一张图片，中间用黑色网格分割
writer = SummaryWriter('./data/tensorboard')
# create grid of images
img_grid = torchvision.utils.make_grid(images)
writer.add_image('image_grid', img_grid)
writer.close()

# 将多张图片直接写入
writer = SummaryWriter('./data/tensorboard')
writer.add_images("images",images,global_step = 0)
writer.close()
```

```
{'0_airplane': 0, '1_automobile': 1}
```



五，可视化人工绘图

如果我们将matplotlib绘图的结果在 tensorboard中展示，可以使用 add_figure.

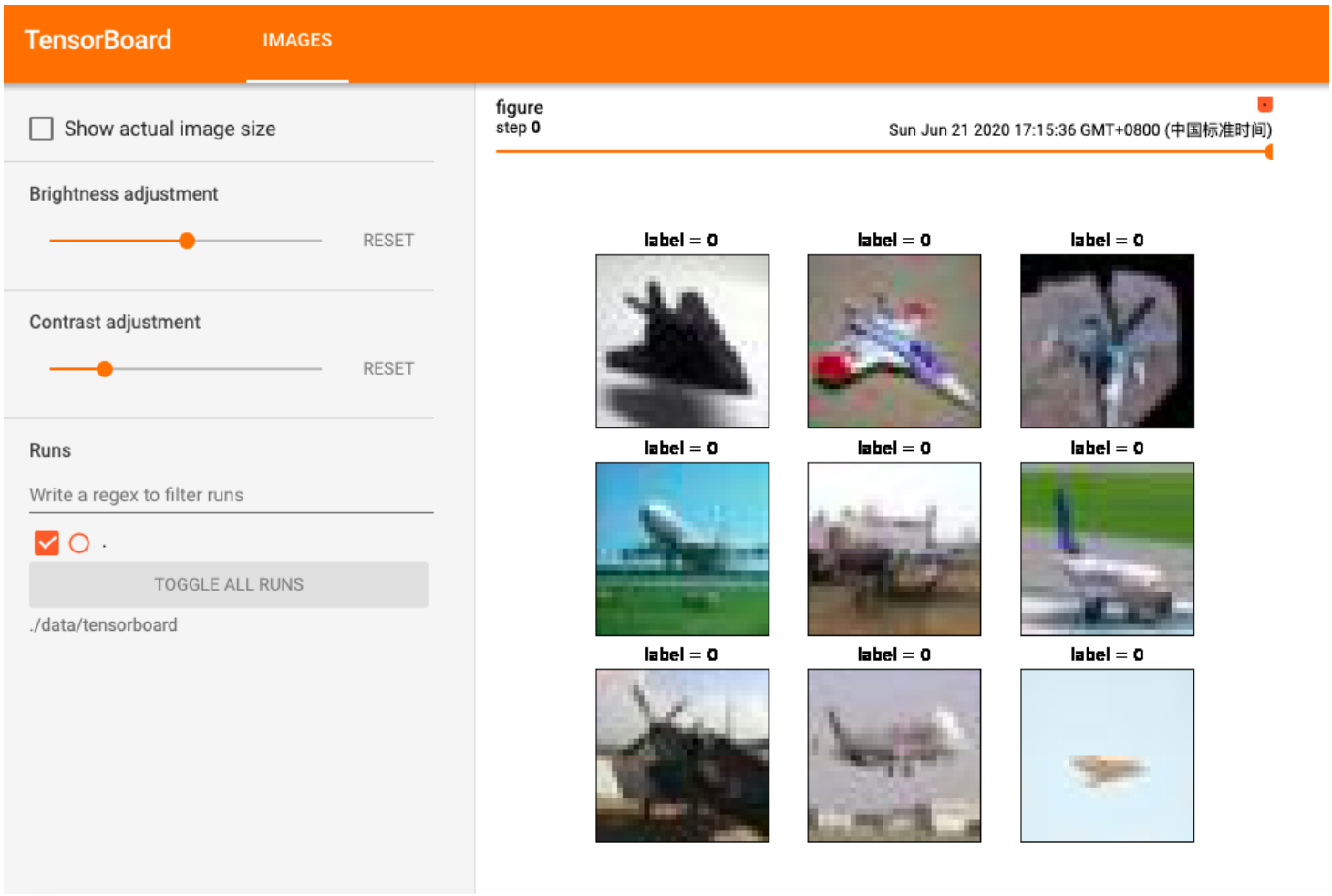
注意，和writer.add_image不同的是，writer.add_figure需要传入matplotlib的figure对象。

```
import torch
import torchvision
from torch import nn
from torch.utils.data import Dataset,DataLoader
from torchvision import transforms,datasets


transform_train = transforms.Compose(
    [transforms.ToTensor()])
transform_valid = transforms.Compose(
    [transforms.ToTensor()])

ds_train = datasets.ImageFolder("./data/cifar2/train/",
            transform = transform_train,target_transform= lambda t:torch.tensor([t]).float())
ds_valid = datasets.ImageFolder("./data/cifar2/test/",
            transform = transform_train,target_transform= lambda t:torch.tensor([t]).float())

print(ds_train.class_to_idx)
```

```
{'0_airplane': 0, '1_automobile': 1}
```

```
%matplotlib inline
%config InlineBackend.figure_format = 'svg'
from matplotlib import pyplot as plt

figure = plt.figure(figsize=(8,8))
for i in range(9):
    img,label = ds_train[i]
    img = img.permute(1,2,0)
    ax=plt.subplot(3,3,i+1)
    ax.imshow(img.numpy())
    ax.set_title("label = %d"%label.item())
    ax.set_xticks([])
    ax.set_yticks([])
plt.show()
```


```
writer = SummaryWriter('./data/tensorboard')
writer.add_figure('figure',figure,global_step=0)
writer.close()
```



如果对本书内容理解上有需要进一步和作者交流的地方，欢迎在公众号"Python与算法之美"下留言。作者时间和精力有限，会酌情予以回复。

也可以在公众号后台回复关键字：**加群**，加入读者交流群和大家讨论。

6-1,构建模型的3种方法

可以使用以下3种方式构建模型：

1，继承nn.Module基类构建自定义模型。

2，使用nn.Sequential按层顺序构建模型。

3，继承nn.Module基类构建模型并辅助应用模型容器进行封装(nn.Sequential,nn.ModuleList,nn.ModuleDict)。

其中 第1种方式最为常见，第2种方式最简单，第3种方式最为灵活也较为复杂。

推荐使用第1种方式构建模型。

```
import torch
from torch import nn
from torchkeras import summary
```

一，继承nn.Module基类构建自定义模型

以下是继承nn.Module基类构建自定义模型的一个范例。模型中的用到的层一般在 `__init__` 函数中定义，然后在 `forward` 方法中定义模型的正向传播逻辑。

```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=3,out_channels=32,kernel_size = 3)
        self.pool1 = nn.MaxPool2d(kernel_size = 2,stride = 2)
        self.conv2 = nn.Conv2d(in_channels=32,out_channels=64,kernel_size = 5)
        self.pool2 = nn.MaxPool2d(kernel_size = 2,stride = 2)
        self.dropout = nn.Dropout2d(p = 0.1)
        self.adaptive_pool = nn.AdaptiveMaxPool2d((1,1))
        self.flatten = nn.Flatten()
        self.linear1 = nn.Linear(64,32)
        self.relu = nn.ReLU()
        self.linear2 = nn.Linear(32,1)
        self.sigmoid = nn.Sigmoid()

    def forward(self,x):
        x = self.conv1(x)
        x = self.pool1(x)
        x = self.conv2(x)
        x = self.pool2(x)
        x = self.dropout(x)
        x = self.adaptive_pool(x)
        x = self.flatten(x)
        x = self.linear1(x)
        x = self.relu(x)
        x = self.linear2(x)
        y = self.sigmoid(x)
        return y

net = Net()
print(net)
```

```
Net(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1))
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1))
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (dropout): Dropout2d(p=0.1, inplace=False)
  (adaptive_pool): AdaptiveMaxPool2d(output_size=(1, 1))
  (flatten): Flatten()
  (linear1): Linear(in_features=64, out_features=32, bias=True)
  (relu): ReLU()
  (linear2): Linear(in_features=32, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)
```

```
summary(net,input_shape= (3,32,32))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 32, 30, 30]             896
         MaxPool2d-2          [-1, 32, 15, 15]               0
            Conv2d-3          [-1, 64, 11, 11]          51,264
         MaxPool2d-4            [-1, 64, 5, 5]               0
         Dropout2d-5            [-1, 64, 5, 5]               0
 AdaptiveMaxPool2d-6            [-1, 64, 1, 1]               0
           Flatten-7                  [-1, 64]               0
            Linear-8                  [-1, 32]           2,080
              ReLU-9                  [-1, 32]               0
           Linear-10                   [-1, 1]              33
          Sigmoid-11                   [-1, 1]               0
================================================================
Total params: 54,273
Trainable params: 54,273
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.011719
Forward/backward pass size (MB): 0.359634
Params size (MB): 0.207035
Estimated Total Size (MB): 0.578388
----------------------------------------------------------------
```

二，使用nn.Sequential按层顺序构建模型

使用nn.Sequential按层顺序构建模型无需定义forward方法。仅仅适合于简单的模型。

以下是使用nn.Sequential搭建模型的一些等价方法。

1，利用add_module方法

```
net = nn.Sequential()
net.add_module("conv1",nn.Conv2d(in_channels=3,out_channels=32,kernel_size = 3))
net.add_module("pool1",nn.MaxPool2d(kernel_size = 2,stride = 2))
net.add_module("conv2",nn.Conv2d(in_channels=32,out_channels=64,kernel_size = 5))
net.add_module("pool2",nn.MaxPool2d(kernel_size = 2,stride = 2))
net.add_module("dropout",nn.Dropout2d(p = 0.1))
net.add_module("adaptive_pool",nn.AdaptiveMaxPool2d((1,1)))
net.add_module("flatten",nn.Flatten())
net.add_module("linear1",nn.Linear(64,32))
net.add_module("relu",nn.ReLU())
net.add_module("linear2",nn.Linear(32,1))
net.add_module("sigmoid",nn.Sigmoid())

print(net)
```

```
Sequential(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1))
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1))
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (dropout): Dropout2d(p=0.1, inplace=False)
  (adaptive_pool): AdaptiveMaxPool2d(output_size=(1, 1))
  (flatten): Flatten()
  (linear1): Linear(in_features=64, out_features=32, bias=True)
  (relu): ReLU()
  (linear2): Linear(in_features=32, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)
```

2，利用变长参数

这种方式构建时不能给每个层指定名称。

```
net = nn.Sequential(
    nn.Conv2d(in_channels=3,out_channels=32,kernel_size = 3),
    nn.MaxPool2d(kernel_size = 2,stride = 2),
    nn.Conv2d(in_channels=32,out_channels=64,kernel_size = 5),
    nn.MaxPool2d(kernel_size = 2,stride = 2),
    nn.Dropout2d(p = 0.1),
    nn.AdaptiveMaxPool2d((1,1)),
    nn.Flatten(),
    nn.Linear(64,32),
    nn.ReLU(),
    nn.Linear(32,1),
    nn.Sigmoid()
)

print(net)
```

```
Sequential(
  (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1))
  (1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (2): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1))
  (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (4): Dropout2d(p=0.1, inplace=False)
  (5): AdaptiveMaxPool2d(output_size=(1, 1))
  (6): Flatten()
  (7): Linear(in_features=64, out_features=32, bias=True)
  (8): ReLU()
  (9): Linear(in_features=32, out_features=1, bias=True)
  (10): Sigmoid()
)
```

3，利用OrderedDict

```
from collections import OrderedDict

net = nn.Sequential(OrderedDict(
          [("conv1",nn.Conv2d(in_channels=3,out_channels=32,kernel_size = 3)),
            ("pool1",nn.MaxPool2d(kernel_size = 2,stride = 2)),
            ("conv2",nn.Conv2d(in_channels=32,out_channels=64,kernel_size = 5)),
            ("pool2",nn.MaxPool2d(kernel_size = 2,stride = 2)),
            ("dropout",nn.Dropout2d(p = 0.1)),
            ("adaptive_pool",nn.AdaptiveMaxPool2d((1,1))),
            ("flatten",nn.Flatten()),
            ("linear1",nn.Linear(64,32)),
            ("relu",nn.ReLU()),
            ("linear2",nn.Linear(32,1)),
            ("sigmoid",nn.Sigmoid())
          ])
        )
print(net)
```

```
Sequential(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1))
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1))
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (dropout): Dropout2d(p=0.1, inplace=False)
  (adaptive_pool): AdaptiveMaxPool2d(output_size=(1, 1))
  (flatten): Flatten()
  (linear1): Linear(in_features=64, out_features=32, bias=True)
  (relu): ReLU()
  (linear2): Linear(in_features=32, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)
```

```
summary(net,input_shape= (3,32,32))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 30, 30]             896
         MaxPool2d-2           [-1, 32, 15, 15]               0
            Conv2d-3           [-1, 64, 11, 11]          51,264
         MaxPool2d-4             [-1, 64, 5, 5]               0
         Dropout2d-5             [-1, 64, 5, 5]               0
 AdaptiveMaxPool2d-6             [-1, 64, 1, 1]               0
           Flatten-7                   [-1, 64]               0
            Linear-8                   [-1, 32]           2,080
              ReLU-9                   [-1, 32]               0
           Linear-10                    [-1, 1]              33
          Sigmoid-11                    [-1, 1]               0
================================================================
Total params: 54,273
Trainable params: 54,273
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.011719
Forward/backward pass size (MB): 0.359634
Params size (MB): 0.207035
Estimated Total Size (MB): 0.578388
----------------------------------------------------------------
```

三，继承nn.Module基类构建模型并辅助应用模型容器进行封装

当模型的结构比较复杂时，我们可以应用模型容器(nn.Sequential,nn.ModuleList,nn.ModuleDict)对模型的部分结构进行封装。

这样做会让模型整体更加有层次感，有时候也能减少代码量。

注意，在下面的范例中我们每次仅仅使用一种模型容器，但实际上这些模型容器的使用是非常灵活的，可以在一个模型中任意组合任意嵌套使用。

1，nn.Sequential作为模型容器

```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_channels=3,out_channels=32,kernel_size = 3),
            nn.MaxPool2d(kernel_size = 2,stride = 2),
            nn.Conv2d(in_channels=32,out_channels=64,kernel_size = 5),
            nn.MaxPool2d(kernel_size = 2,stride = 2),
            nn.Dropout2d(p = 0.1),
            nn.AdaptiveMaxPool2d((1,1))
        )
        self.dense = nn.Sequential(
            nn.Flatten(),
            nn.Linear(64,32),
            nn.ReLU(),
            nn.Linear(32,1),
            nn.Sigmoid()
        )
    def forward(self,x):
        x = self.conv(x)
        y = self.dense(x)
        return y

net = Net()
print(net)
```

```
Net(
  (conv): Sequential(
    (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1))
    (1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (2): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1))
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (4): Dropout2d(p=0.1, inplace=False)
    (5): AdaptiveMaxPool2d(output_size=(1, 1))
  )
  (dense): Sequential(
    (0): Flatten()
    (1): Linear(in_features=64, out_features=32, bias=True)
    (2): ReLU()
    (3): Linear(in_features=32, out_features=1, bias=True)
    (4): Sigmoid()
  )
)
```

2，nn.ModuleList作为模型容器

注意下面中的ModuleList不能用Python中的列表代替。

```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.layers = nn.ModuleList([
            nn.Conv2d(in_channels=3,out_channels=32,kernel_size = 3),
            nn.MaxPool2d(kernel_size = 2,stride = 2),
            nn.Conv2d(in_channels=32,out_channels=64,kernel_size = 5),
            nn.MaxPool2d(kernel_size = 2,stride = 2),
            nn.Dropout2d(p = 0.1),
            nn.AdaptiveMaxPool2d((1,1)),
            nn.Flatten(),
            nn.Linear(64,32),
            nn.ReLU(),
            nn.Linear(32,1),
            nn.Sigmoid()]
        )
    def forward(self,x):
        for layer in self.layers:
            x = layer(x)
        return x
net = Net()
print(net)
```

```
Net(
  (layers): ModuleList(
    (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1))
    (1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (2): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1))
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (4): Dropout2d(p=0.1, inplace=False)
    (5): AdaptiveMaxPool2d(output_size=(1, 1))
    (6): Flatten()
    (7): Linear(in_features=64, out_features=32, bias=True)
    (8): ReLU()
    (9): Linear(in_features=32, out_features=1, bias=True)
    (10): Sigmoid()
  )
)
```

```
summary(net,input_shape= (3,32,32))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 30, 30]             896
         MaxPool2d-2           [-1, 32, 15, 15]               0
            Conv2d-3           [-1, 64, 11, 11]          51,264
         MaxPool2d-4             [-1, 64, 5, 5]               0
         Dropout2d-5             [-1, 64, 5, 5]               0
 AdaptiveMaxPool2d-6             [-1, 64, 1, 1]               0
           Flatten-7                   [-1, 64]               0
            Linear-8                   [-1, 32]           2,080
              ReLU-9                   [-1, 32]               0
           Linear-10                    [-1, 1]              33
          Sigmoid-11                    [-1, 1]               0
================================================================
Total params: 54,273
Trainable params: 54,273
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.011719
Forward/backward pass size (MB): 0.359634
Params size (MB): 0.207035
Estimated Total Size (MB): 0.578388
----------------------------------------------------------------
```

3，nn.ModuleDict作为模型容器

注意下面中的ModuleDict不能用Python中的字典代替。

```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.layers_dict = nn.ModuleDict({"conv1":nn.Conv2d(in_channels=3,out_channels=32,kernel
               "pool": nn.MaxPool2d(kernel_size = 2,stride = 2),
               "conv2":nn.Conv2d(in_channels=32,out_channels=64,kernel_size = 5),
               "dropout": nn.Dropout2d(p = 0.1),
               "adaptive":nn.AdaptiveMaxPool2d((1,1)),
               "flatten": nn.Flatten(),
               "linear1": nn.Linear(64,32),
               "relu":nn.ReLU(),
               "linear2": nn.Linear(32,1),
               "sigmoid": nn.Sigmoid()
              })
    def forward(self,x):
        layers = ["conv1","pool","conv2","pool","dropout","adaptive",
                  "flatten","linear1","relu","linear2","sigmoid"]
        for layer in layers:
            x = self.layers_dict[layer](x)
        return x
net = Net()
print(net)
```

```
Net(
  (layers_dict): ModuleDict(
    (adaptive): AdaptiveMaxPool2d(output_size=(1, 1))
    (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1))
    (conv2): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1))
    (dropout): Dropout2d(p=0.1, inplace=False)
    (flatten): Flatten()
    (linear1): Linear(in_features=64, out_features=32, bias=True)
    (linear2): Linear(in_features=32, out_features=1, bias=True)
    (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (relu): ReLU()
    (sigmoid): Sigmoid()
  )
)
```

```
summary(net,input_shape= (3,32,32))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 32, 30, 30]             896
         MaxPool2d-2          [-1, 32, 15, 15]               0
            Conv2d-3          [-1, 64, 11, 11]          51,264
         MaxPool2d-4            [-1, 64, 5, 5]               0
         Dropout2d-5            [-1, 64, 5, 5]               0
 AdaptiveMaxPool2d-6            [-1, 64, 1, 1]               0
           Flatten-7                  [-1, 64]               0
            Linear-8                  [-1, 32]           2,080
              ReLU-9                  [-1, 32]               0
           Linear-10                   [-1, 1]              33
          Sigmoid-11                   [-1, 1]               0
================================================================
Total params: 54,273
Trainable params: 54,273
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.011719
Forward/backward pass size (MB): 0.359634
Params size (MB): 0.207035
Estimated Total Size (MB): 0.578388
----------------------------------------------------------------
```

如果对本书内容理解上有需要进一步和作者交流的地方，欢迎在公众号"Python与算法之美"下留言。作者时间和精力有限，会酌情予以回复。

也可以在公众号后台回复关键字：**加群**，加入读者交流群和大家讨论。

6-2,训练模型的3种方法

Pytorch通常需要用户编写自定义训练循环，训练循环的代码风格因人而异。

有3类典型的训练循环代码风格：脚本形式训练循环，函数形式训练循环，类形式训练循环。

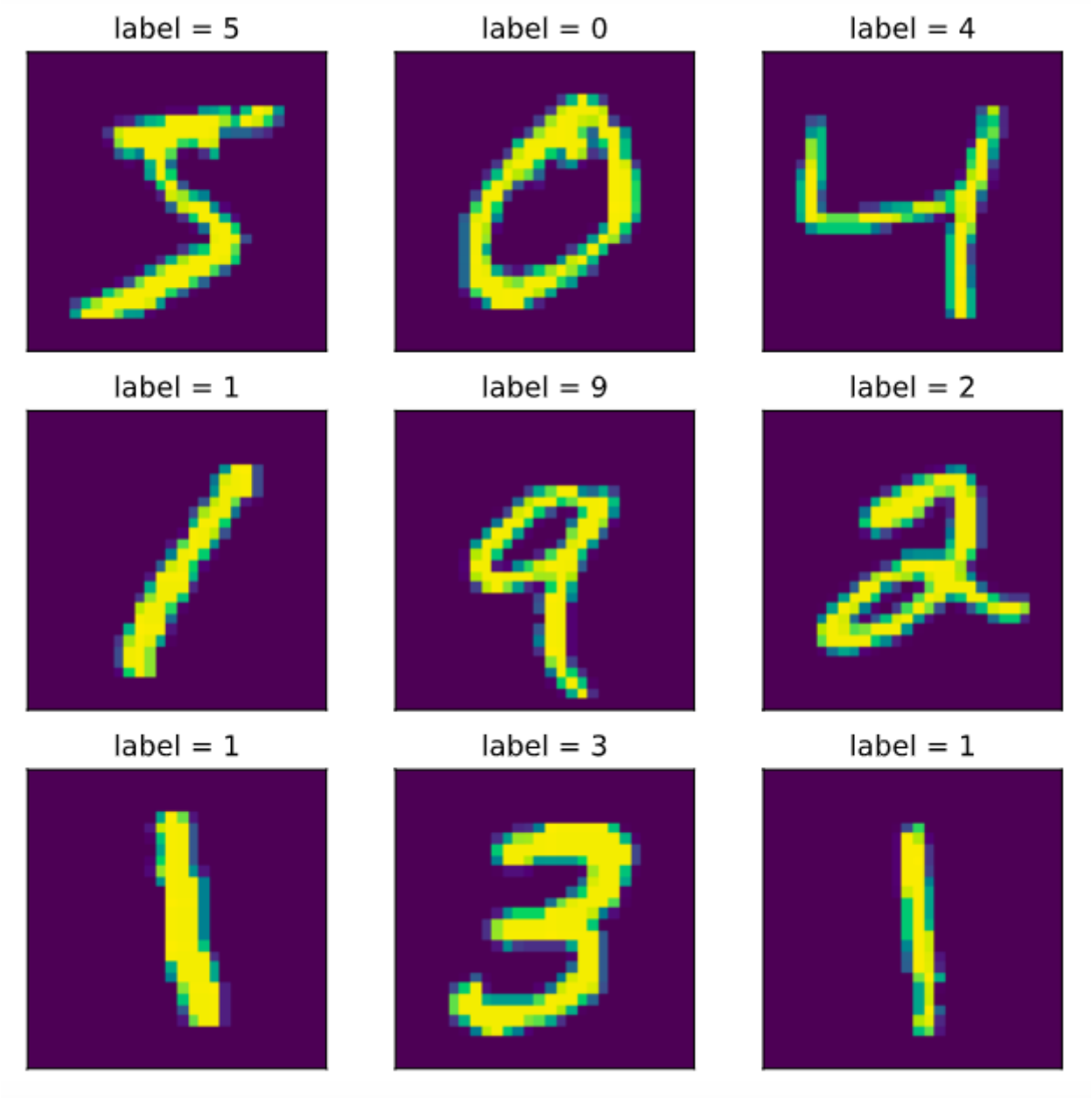
下面以minist数据集的分类模型的训练为例，演示这3种训练模型的风格。

〇，准备数据

```
import torch
from torch import nn
from torchkeras import summary,Model

import torchvision
from torchvision import transforms



transform = transforms.Compose([transforms.ToTensor()])

ds_train = torchvision.datasets.MNIST(root="./data/minist/",train=True,download=True,transform=t
ds_valid = torchvision.datasets.MNIST(root="./data/minist/",train=False,download=True,transform=

dl_train =  torch.utils.data.DataLoader(ds_train, batch_size=128, shuffle=True, num_workers=4)
dl_valid =  torch.utils.data.DataLoader(ds_valid, batch_size=128, shuffle=False, num_workers=4)

print(len(ds_train))
print(len(ds_valid))
```

```
60000
10000
```

```
%matplotlib inline
%config InlineBackend.figure_format = 'svg'

#查看部分样本
from matplotlib import pyplot as plt

plt.figure(figsize=(8,8))
for i in range(9):
    img,label = ds_train[i]
    img = torch.squeeze(img)
    ax=plt.subplot(3,3,i+1)
    ax.imshow(img.numpy())
    ax.set_title("label = %d"%label)
    ax.set_xticks([])
    ax.set_yticks([])
plt.show()
```



一，脚本风格

脚本风格的训练循环最为常见。

```
net = nn.Sequential()
net.add_module("conv1",nn.Conv2d(in_channels=1,out_channels=32,kernel_size = 3))
net.add_module("pool1",nn.MaxPool2d(kernel_size = 2,stride = 2))
net.add_module("conv2",nn.Conv2d(in_channels=32,out_channels=64,kernel_size = 5))
net.add_module("pool2",nn.MaxPool2d(kernel_size = 2,stride = 2))
net.add_module("dropout",nn.Dropout2d(p = 0.1))
net.add_module("adaptive_pool",nn.AdaptiveMaxPool2d((1,1)))
net.add_module("flatten",nn.Flatten())
net.add_module("linear1",nn.Linear(64,32))
net.add_module("relu",nn.ReLU())
net.add_module("linear2",nn.Linear(32,10))

print(net)
```

```
Sequential(
  (conv1): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1))
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1))
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (dropout): Dropout2d(p=0.1, inplace=False)
  (adaptive_pool): AdaptiveMaxPool2d(output_size=(1, 1))
  (flatten): Flatten()
  (linear1): Linear(in_features=64, out_features=32, bias=True)
  (relu): ReLU()
  (linear2): Linear(in_features=32, out_features=10, bias=True)
)
```

```
summary(net,input_shape=(1,32,32))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 32, 30, 30]             320
         MaxPool2d-2          [-1, 32, 15, 15]               0
            Conv2d-3          [-1, 64, 11, 11]          51,264
         MaxPool2d-4            [-1, 64, 5, 5]               0
         Dropout2d-5            [-1, 64, 5, 5]               0
 AdaptiveMaxPool2d-6            [-1, 64, 1, 1]               0
           Flatten-7                  [-1, 64]               0
            Linear-8                  [-1, 32]           2,080
              ReLU-9                  [-1, 32]               0
           Linear-10                  [-1, 10]             330
================================================================
Total params: 53,994
Trainable params: 53,994
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.003906
Forward/backward pass size (MB): 0.359695
Params size (MB): 0.205971
Estimated Total Size (MB): 0.569572
----------------------------------------------------------------
```

```
import datetime
import numpy as np
import pandas as pd
from sklearn.metrics import accuracy_score

def accuracy(y_pred,y_true):
    y_pred_cls = torch.argmax(nn.Softmax(dim=1)(y_pred),dim=1).data
    return accuracy_score(y_true,y_pred_cls)

loss_func = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(params=net.parameters(),lr = 0.01)
metric_func = accuracy
metric_name = "accuracy"
```

```
epochs = 3
log_step_freq = 100

dfhistory = pd.DataFrame(columns = ["epoch","loss",metric_name,"val_loss","val_"+metric_name])
print("Start Training...")
nowtime = datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S')
print("==========="*8 + "%s"%nowtime)

for epoch in range(1,epochs+1):

    # 1，训练循环-------------------------------------------------
    net.train()
    loss_sum = 0.0
    metric_sum = 0.0
    step = 1

    for step, (features,labels) in enumerate(dl_train, 1):

        # 梯度清零
        optimizer.zero_grad()

        # 正向传播求损失
        predictions = net(features)
        loss = loss_func(predictions,labels)
        metric = metric_func(predictions,labels)

        # 反向传播求梯度
        loss.backward()
        optimizer.step()

        # 打印batch级别日志
        loss_sum += loss.item()
        metric_sum += metric.item()
        if step%log_step_freq == 0:
            print(("[step = %d] loss: %.3f, "+metric_name+": %.3f") %
                  (step, loss_sum/step, metric_sum/step))

    # 2，验证循环-------------------------------------------------
    net.eval()
    val_loss_sum = 0.0
    val_metric_sum = 0.0
    val_step = 1

    for val_step, (features,labels) in enumerate(dl_valid, 1):
        with torch.no_grad():
            predictions = net(features)
            val_loss = loss_func(predictions,labels)
            val_metric = metric_func(predictions,labels)

        val_loss_sum += val_loss.item()
        val_metric_sum += val_metric.item()
```

```
    # 3，记录日志-----------------------------------------------
    info = (epoch, loss_sum/step, metric_sum/step,
            val_loss_sum/val_step, val_metric_sum/val_step)
    dfhistory.loc[epoch-1] = info

    # 打印epoch级别日志
    print(("\nEPOCH = %d, loss = %.3f,"+ metric_name + \
          "  = %.3f, val_loss = %.3f, "+"val_"+ metric_name+" = %.3f")
          %info)
    nowtime = datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S')
    print("\n"+"=========="*8 + "%s"%nowtime)

print('Finished Training...')
```

```
Start Training...
================================================================================2020-06-26 12:49:16
[step = 100] loss: 0.742, accuracy: 0.745
[step = 200] loss: 0.466, accuracy: 0.843
[step = 300] loss: 0.363, accuracy: 0.880
[step = 400] loss: 0.310, accuracy: 0.898

EPOCH = 1, loss = 0.281,accuracy  = 0.908, val_loss = 0.087, val_accuracy = 0.972

================================================================================2020-06-26 12:50:32
[step = 100] loss: 0.103, accuracy: 0.970
[step = 200] loss: 0.114, accuracy: 0.966
[step = 300] loss: 0.112, accuracy: 0.967
[step = 400] loss: 0.108, accuracy: 0.968

EPOCH = 2, loss = 0.111,accuracy  = 0.967, val_loss = 0.082, val_accuracy = 0.976

================================================================================2020-06-26 12:51:47
[step = 100] loss: 0.093, accuracy: 0.972
[step = 200] loss: 0.095, accuracy: 0.971
[step = 300] loss: 0.092, accuracy: 0.972
[step = 400] loss: 0.093, accuracy: 0.972

EPOCH = 3, loss = 0.098,accuracy  = 0.971, val_loss = 0.113, val_accuracy = 0.970

================================================================================2020-06-26 12:53:09
Finished Training...
```

二，函数风格

该风格在脚本形式上作了简单的函数封装。

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.layers = nn.ModuleList([
            nn.Conv2d(in_channels=1,out_channels=32,kernel_size = 3),
            nn.MaxPool2d(kernel_size = 2,stride = 2),
            nn.Conv2d(in_channels=32,out_channels=64,kernel_size = 5),
            nn.MaxPool2d(kernel_size = 2,stride = 2),
            nn.Dropout2d(p = 0.1),
            nn.AdaptiveMaxPool2d((1,1)),
            nn.Flatten(),
            nn.Linear(64,32),
            nn.ReLU(),
            nn.Linear(32,10)]
        )
    def forward(self,x):
        for layer in self.layers:
            x = layer(x)
        return x
net = Net()
print(net)
```

```
Net(
  (layers): ModuleList(
    (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1))
    (1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (2): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1))
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (4): Dropout2d(p=0.1, inplace=False)
    (5): AdaptiveMaxPool2d(output_size=(1, 1))
    (6): Flatten()
    (7): Linear(in_features=64, out_features=32, bias=True)
    (8): ReLU()
    (9): Linear(in_features=32, out_features=10, bias=True)
  )
)
```

```python
summary(net,input_shape=(1,32,32))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 32, 30, 30]             320
         MaxPool2d-2          [-1, 32, 15, 15]               0
            Conv2d-3          [-1, 64, 11, 11]          51,264
         MaxPool2d-4            [-1, 64, 5, 5]               0
         Dropout2d-5            [-1, 64, 5, 5]               0
 AdaptiveMaxPool2d-6            [-1, 64, 1, 1]               0
           Flatten-7                  [-1, 64]               0
            Linear-8                  [-1, 32]           2,080
              ReLU-9                  [-1, 32]               0
           Linear-10                  [-1, 10]             330
================================================================
Total params: 53,994
Trainable params: 53,994
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.003906
Forward/backward pass size (MB): 0.359695
Params size (MB): 0.205971
Estimated Total Size (MB): 0.569572
----------------------------------------------------------------
```

```
import datetime
import numpy as np
import pandas as pd
from sklearn.metrics import accuracy_score

def accuracy(y_pred,y_true):
    y_pred_cls = torch.argmax(nn.Softmax(dim=1)(y_pred),dim=1).data
    return accuracy_score(y_true,y_pred_cls)

model = net
model.optimizer = torch.optim.SGD(model.parameters(),lr = 0.01)
model.loss_func = nn.CrossEntropyLoss()
model.metric_func = accuracy
model.metric_name = "accuracy"
```

```
def train_step(model,features,labels):

    # 训练模式，dropout层发生作用
    model.train()

    # 梯度清零
    model.optimizer.zero_grad()

    # 正向传播求损失
    predictions = model(features)
    loss = model.loss_func(predictions,labels)
    metric = model.metric_func(predictions,labels)

    # 反向传播求梯度
    loss.backward()
    model.optimizer.step()

    return loss.item(),metric.item()

@torch.no_grad()
def valid_step(model,features,labels):

    # 预测模式，dropout层不发生作用
    model.eval()

    predictions = model(features)
    loss = model.loss_func(predictions,labels)
    metric = model.metric_func(predictions,labels)

    return loss.item(), metric.item()


# 测试train_step效果
features,labels = next(iter(dl_train))
train_step(model,features,labels)
```

```
(2.32741117477417, 0.1015625)
```

```
def train_model(model,epochs,dl_train,dl_valid,log_step_freq):

    metric_name = model.metric_name
    dfhistory = pd.DataFrame(columns = ["epoch","loss",metric_name,"val_loss","val_"+metric_name
    print("Start Training...")
    nowtime = datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S')
    print("=========="*8 + "%s"%nowtime)

    for epoch in range(1,epochs+1):

        # 1，训练循环-------------------------------------------------
        loss_sum = 0.0
        metric_sum = 0.0
        step = 1

        for step, (features,labels) in enumerate(dl_train, 1):

            loss,metric = train_step(model,features,labels)

            # 打印batch级别日志
            loss_sum += loss
            metric_sum += metric
            if step%log_step_freq == 0:
                print(("[step = %d] loss: %.3f, "+metric_name+": %.3f") %
                      (step, loss_sum/step, metric_sum/step))

        # 2，验证循环-------------------------------------------------
        val_loss_sum = 0.0
        val_metric_sum = 0.0
        val_step = 1

        for val_step, (features,labels) in enumerate(dl_valid, 1):

            val_loss,val_metric = valid_step(model,features,labels)

            val_loss_sum += val_loss
            val_metric_sum += val_metric

        # 3，记录日志-------------------------------------------------
        info = (epoch, loss_sum/step, metric_sum/step,
                val_loss_sum/val_step, val_metric_sum/val_step)
        dfhistory.loc[epoch-1] = info

        # 打印epoch级别日志
        print(("\nEPOCH = %d, loss = %.3f,"+ metric_name + \
              "  = %.3f, val_loss = %.3f, "+"val_"+ metric_name+" = %.3f")
              %info)
        nowtime = datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S')
        print("\n"+"=========="*8 + "%s"%nowtime)

    print('Finished Training...')
```

```
    return dfhistory
```

```
epochs = 3
dfhistory = train_model(model,epochs,dl_train,dl_valid,log_step_freq = 100)
```

```
Start Training...
================================================================================2020-06-26 13:10:00
[step = 100] loss: 2.298, accuracy: 0.137
[step = 200] loss: 2.288, accuracy: 0.145
[step = 300] loss: 2.278, accuracy: 0.165
[step = 400] loss: 2.265, accuracy: 0.183

EPOCH = 1, loss = 2.254,accuracy  = 0.195, val_loss = 2.158, val_accuracy = 0.301

================================================================================2020-06-26 13:11:23
[step = 100] loss: 2.127, accuracy: 0.302
[step = 200] loss: 2.080, accuracy: 0.338
[step = 300] loss: 2.025, accuracy: 0.374
[step = 400] loss: 1.957, accuracy: 0.411

EPOCH = 2, loss = 1.905,accuracy  = 0.435, val_loss = 1.469, val_accuracy = 0.710

================================================================================2020-06-26 13:12:43
[step = 100] loss: 1.435, accuracy: 0.615
[step = 200] loss: 1.324, accuracy: 0.647
[step = 300] loss: 1.221, accuracy: 0.672
[step = 400] loss: 1.132, accuracy: 0.696

EPOCH = 3, loss = 1.074,accuracy  = 0.711, val_loss = 0.582, val_accuracy = 0.878

================================================================================2020-06-26 13:13:59
Finished Training...
```

三，类风格

此处使用torchkeras中定义的模型接口构建模型，并调用compile方法和fit方法训练模型。

使用该形式训练模型非常简洁明了。推荐使用该形式。

```
class CnnModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = nn.ModuleList([
            nn.Conv2d(in_channels=1,out_channels=32,kernel_size = 3),
            nn.MaxPool2d(kernel_size = 2,stride = 2),
            nn.Conv2d(in_channels=32,out_channels=64,kernel_size = 5),
            nn.MaxPool2d(kernel_size = 2,stride = 2),
            nn.Dropout2d(p = 0.1),
            nn.AdaptiveMaxPool2d((1,1)),
            nn.Flatten(),
            nn.Linear(64,32),
            nn.ReLU(),
            nn.Linear(32,10)]
        )
    def forward(self,x):
        for layer in self.layers:
            x = layer(x)
        return x
model = torchkeras.Model(CnnModel())
print(model)
```

```
CnnModel(
  (layers): ModuleList(
    (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1))
    (1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (2): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1))
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (4): Dropout2d(p=0.1, inplace=False)
    (5): AdaptiveMaxPool2d(output_size=(1, 1))
    (6): Flatten()
    (7): Linear(in_features=64, out_features=32, bias=True)
    (8): ReLU()
    (9): Linear(in_features=32, out_features=10, bias=True)
  )
)
```

```
model.summary(input_shape=(1,32,32))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 32, 30, 30]             320
         MaxPool2d-2          [-1, 32, 15, 15]               0
            Conv2d-3          [-1, 64, 11, 11]          51,264
         MaxPool2d-4            [-1, 64, 5, 5]               0
         Dropout2d-5            [-1, 64, 5, 5]               0
 AdaptiveMaxPool2d-6            [-1, 64, 1, 1]               0
           Flatten-7                  [-1, 64]               0
            Linear-8                  [-1, 32]           2,080
              ReLU-9                  [-1, 32]               0
           Linear-10                  [-1, 10]             330
================================================================
Total params: 53,994
Trainable params: 53,994
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.003906
Forward/backward pass size (MB): 0.359695
Params size (MB): 0.205971
Estimated Total Size (MB): 0.569572
----------------------------------------------------------------
```

```
from sklearn.metrics import accuracy_score

def accuracy(y_pred,y_true):
    y_pred_cls = torch.argmax(nn.Softmax(dim=1)(y_pred),dim=1).data
    return accuracy_score(y_true.numpy(),y_pred_cls.numpy())

model.compile(loss_func = nn.CrossEntropyLoss(),
              optimizer= torch.optim.Adam(model.parameters(),lr = 0.02),
              metrics_dict={"accuracy":accuracy})



dfhistory = model.fit(3,dl_train = dl_train, dl_val=dl_valid, log_step_freq=100)
```

```
Start Training ...

================================================================================2020-06-26 13:22:39
{'step': 100, 'loss': 0.976, 'accuracy': 0.664}
{'step': 200, 'loss': 0.611, 'accuracy': 0.795}
{'step': 300, 'loss': 0.478, 'accuracy': 0.841}
{'step': 400, 'loss': 0.403, 'accuracy': 0.868}

 +-------+-------+----------+----------+--------------+
| epoch |  loss | accuracy | val_loss | val_accuracy |
+-------+-------+----------+----------+--------------+
|   1   | 0.371 |  0.879   |  0.087   |    0.972     |
+-------+-------+----------+----------+--------------+

================================================================================2020-06-26 13:23:59
{'step': 100, 'loss': 0.182, 'accuracy': 0.948}
{'step': 200, 'loss': 0.176, 'accuracy': 0.949}
{'step': 300, 'loss': 0.173, 'accuracy': 0.95}
{'step': 400, 'loss': 0.174, 'accuracy': 0.951}

 +-------+-------+----------+----------+--------------+
| epoch |  loss | accuracy | val_loss | val_accuracy |
+-------+-------+----------+----------+--------------+
|   2   | 0.175 |  0.951   |  0.152   |    0.958     |
+-------+-------+----------+----------+--------------+

================================================================================2020-06-26 13:25:22
{'step': 100, 'loss': 0.143, 'accuracy': 0.961}
{'step': 200, 'loss': 0.151, 'accuracy': 0.959}
{'step': 300, 'loss': 0.149, 'accuracy': 0.96}
{'step': 400, 'loss': 0.152, 'accuracy': 0.959}

 +-------+-------+----------+----------+--------------+
| epoch |  loss | accuracy | val_loss | val_accuracy |
+-------+-------+----------+----------+--------------+
|   3   | 0.153 |  0.959   |  0.086   |    0.975     |
+-------+-------+----------+----------+--------------+

================================================================================2020-06-26 13:26:48
Finished Training...
```

如果对本书内容理解上有需要进一步和作者交流的地方，欢迎在公众号"Python与算法之美"下留言。作者时间和精力有限，会酌情予以回复。

也可以在公众号后台回复关键字：**加群**，加入读者交流群和大家讨论。

6-3,使用GPU训练模型

深度学习的训练过程常常非常耗时，一个模型训练几个小时是家常便饭，训练几天也是常有的事情，有时候甚至要训练几十天。

训练过程的耗时主要来自于两个部分，一部分来自数据准备，另一部分来自参数迭代。

当数据准备过程还是模型训练时间的主要瓶颈时，我们可以使用更多进程来准备数据。

当参数迭代过程成为训练时间的主要瓶颈时，我们通常的方法是应用GPU来进行加速。

Pytorch中使用GPU加速模型非常简单，只要将模型和数据移动到GPU上。核心代码只有以下几行。

```
# 定义模型
...

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model.to(device) # 移动模型到cuda

# 训练模型
...

features = features.to(device) # 移动数据到cuda
labels = labels.to(device) # 或者  labels = labels.cuda() if torch.cuda.is_available() else labe
...
```

如果要使用多个GPU训练模型，也非常简单。只需要在将模型设置为数据并行风格模型。
则模型移动到GPU上之后，会在每一个GPU上拷贝一个副本，并把数据平分到各个GPU上进行训练。
核心代码如下。

```
# 定义模型
...

if torch.cuda.device_count() > 1:
    model = nn.DataParallel(model) # 包装为并行风格模型

# 训练模型
...
features = features.to(device) # 移动数据到cuda
labels = labels.to(device) # 或者 labels = labels.cuda() if torch.cuda.is_available() else label
...
```

以下是一些和GPU有关的基本操作汇总

在Colab笔记本中：修改->笔记本设置->硬件加速器 中选择 GPU

注：以下代码只能在Colab 上才能正确执行。

可点击如下链接，直接在colab中运行范例代码。

《torch使用gpu训练模型》

https://colab.research.google.com/drive/1FDmi44-U3TFRCt9MwGn4HIj2SaaWIjHu?usp=sharing

```
import torch
from torch import nn


# 1，查看gpu信息
if_cuda = torch.cuda.is_available()
print("if_cuda=",if_cuda)

gpu_count = torch.cuda.device_count()
print("gpu_count=",gpu_count)
```

```
if_cuda= True
gpu_count= 1
```

```
# 2，将张量在gpu和cpu间移动
tensor = torch.rand((100,100))
tensor_gpu = tensor.to("cuda:0") # 或者 tensor_gpu = tensor.cuda()
print(tensor_gpu.device)
print(tensor_gpu.is_cuda)

tensor_cpu = tensor_gpu.to("cpu") # 或者 tensor_cpu = tensor_gpu.cpu()
print(tensor_cpu.device)
```

```
cuda:0
True
cpu
```

```
# 3，将模型中的全部张量移动到gpu上
net = nn.Linear(2,1)
print(next(net.parameters()).is_cuda)
net.to("cuda:0") # 将模型中的全部参数张量依次到GPU上，注意，无需重新赋值为 net = net.to("cuda:0")
print(next(net.parameters()).is_cuda)
print(next(net.parameters()).device)
```

```
False
True
cuda:0
```

```
# 4，创建支持多个gpu数据并行的模型
linear = nn.Linear(2,1)
print(next(linear.parameters()).device)

model = nn.DataParallel(linear)
print(model.device_ids)
print(next(model.module.parameters()).device)

#注意保存参数时要指定保存model.module的参数
torch.save(model.module.state_dict(), "./data/model_parameter.pkl")

linear = nn.Linear(2,1)
linear.load_state_dict(torch.load("./data/model_parameter.pkl"))
```

```
cpu
[0]
cuda:0
```

```
# 5，清空cuda缓存

# 该方法在cuda超内存时十分有用
torch.cuda.empty_cache()
```

一，矩阵乘法范例

下面分别使用CPU和GPU作一个矩阵乘法，并比较其计算效率。

```
import time
import torch
from torch import nn


# 使用cpu
a = torch.rand((10000,200))
b = torch.rand((200,10000))
tic = time.time()
c = torch.matmul(a,b)
toc = time.time()

print(toc-tic)
print(a.device)
print(b.device)
```

```
0.6454010009765625
cpu
cpu
```

```
# 使用gpu
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
a = torch.rand((10000,200),device = device) #可以指定在GPU上创建张量
b = torch.rand((200,10000)) #也可以在CPU上创建张量后移动到GPU上
b = b.to(device) #或者 b = b.cuda() if torch.cuda.is_available() else b
tic = time.time()
c = torch.matmul(a,b)
toc = time.time()
print(toc-tic)
print(a.device)
print(b.device)
```

```
0.014541149139404297
cuda:0
cuda:0
```

二，线性回归范例

下面对比使用CPU和GPU训练一个线性回归模型的效率

1，使用CPU

```
# 准备数据
n = 1000000 #样本数量

X = 10*torch.rand([n,2])-5.0  #torch.rand是均匀分布
w0 = torch.tensor([[2.0,-3.0]])
b0 = torch.tensor([[10.0]])
Y = X@w0.t() + b0 + torch.normal( 0.0,2.0,size = [n,1])  # @表示矩阵乘法,增加正态扰动


# 定义模型
class LinearRegression(nn.Module):
    def __init__(self):
        super().__init__()
        self.w = nn.Parameter(torch.randn_like(w0))
        self.b = nn.Parameter(torch.zeros_like(b0))
    #正向传播
    def forward(self,x):
        return x@self.w.t() + self.b

linear = LinearRegression()
```

```
# 训练模型
optimizer = torch.optim.Adam(linear.parameters(),lr = 0.1)
loss_func = nn.MSELoss()

def train(epoches):
    tic = time.time()
    for epoch in range(epoches):
        optimizer.zero_grad()
        Y_pred = linear(X)
        loss = loss_func(Y_pred,Y)
        loss.backward()
        optimizer.step()
        if epoch%50==0:
            print({"epoch":epoch,"loss":loss.item()})
    toc = time.time()
    print("time used:",toc-tic)

train(500)
```

```
{'epoch': 0, 'loss': 3.996487855911255}
{'epoch': 50, 'loss': 3.9969770908355713}
{'epoch': 100, 'loss': 3.9964890480041504}
{'epoch': 150, 'loss': 3.996488332748413}
{'epoch': 200, 'loss': 3.996488094329834}
{'epoch': 250, 'loss': 3.996488332748413}
{'epoch': 300, 'loss': 3.996488332748413}
{'epoch': 350, 'loss': 3.996488094329834}
{'epoch': 400, 'loss': 3.996488332748413}
{'epoch': 450, 'loss': 3.996488094329834}
time used: 5.4090576171875
```

2，使用GPU

```
# 准备数据
n = 1000000 #样本数量

X = 10*torch.rand([n,2])-5.0  #torch.rand是均匀分布
w0 = torch.tensor([[2.0,-3.0]])
b0 = torch.tensor([[10.0]])
Y = X@w0.t() + b0 + torch.normal( 0.0,2.0,size = [n,1])  # @表示矩阵乘法,增加正态扰动

# 移动到GPU上
print("torch.cuda.is_available() = ",torch.cuda.is_available())
X = X.cuda()
Y = Y.cuda()
print("X.device:",X.device)
print("Y.device:",Y.device)
```

```
torch.cuda.is_available() =  True
X.device: cuda:0
Y.device: cuda:0
```

```
# 定义模型
class LinearRegression(nn.Module):
    def __init__(self):
        super().__init__()
        self.w = nn.Parameter(torch.randn_like(w0))
        self.b = nn.Parameter(torch.zeros_like(b0))
    #正向传播
    def forward(self,x):
        return x@self.w.t() + self.b

linear = LinearRegression()

# 移动模型到GPU上
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
linear.to(device)

#查看模型是否已经移动到GPU上
print("if on cuda:",next(linear.parameters()).is_cuda)
```

```
if on cuda: True
```

```
# 训练模型
optimizer = torch.optim.Adam(linear.parameters(),lr = 0.1)
loss_func = nn.MSELoss()

def train(epoches):
    tic = time.time()
    for epoch in range(epoches):
        optimizer.zero_grad()
        Y_pred = linear(X)
        loss = loss_func(Y_pred,Y)
        loss.backward()
        optimizer.step()
        if epoch%50==0:
            print({"epoch":epoch,"loss":loss.item()})
    toc = time.time()
    print("time used:",toc-tic)

train(500)
```

```
{'epoch': 0, 'loss': 3.9982845783233643}
{'epoch': 50, 'loss': 3.998818874359131}
{'epoch': 100, 'loss': 3.9982895851135254}
{'epoch': 150, 'loss': 3.9982845783233643}
{'epoch': 200, 'loss': 3.998284339904785}
{'epoch': 250, 'loss': 3.9982845783233643}
{'epoch': 300, 'loss': 3.9982845783233643}
{'epoch': 350, 'loss': 3.9982845783233643}
{'epoch': 400, 'loss': 3.9982845783233643}
{'epoch': 450, 'loss': 3.9982845783233643}
time used: 0.4889392852783203
```

三，torchkeras使用单GPU范例

下面演示使用torchkeras来应用GPU训练模型的方法。

其对应的CPU训练模型代码参见《6-2,训练模型的3种方法》

本例仅需要在它的基础上增加一行代码，在model.compile时指定 device即可。

1，准备数据

```
!pip install -U torchkeras
```

```
import torch
from torch import nn

import torchvision
from torchvision import transforms

import torchkeras
```

```
transform = transforms.Compose([transforms.ToTensor()])

ds_train = torchvision.datasets.MNIST(root="./data/minist/",train=True,download=True,transform=t
ds_valid = torchvision.datasets.MNIST(root="./data/minist/",train=False,download=True,transform=

dl_train =  torch.utils.data.DataLoader(ds_train, batch_size=128, shuffle=True, num_workers=4)
dl_valid =  torch.utils.data.DataLoader(ds_valid, batch_size=128, shuffle=False, num_workers=4)

print(len(ds_train))
print(len(ds_valid))
```

```
%matplotlib inline
%config InlineBackend.figure_format = 'svg'

#查看部分样本
from matplotlib import pyplot as plt

plt.figure(figsize=(8,8))
for i in range(9):
    img,label = ds_train[i]
    img = torch.squeeze(img)
    ax=plt.subplot(3,3,i+1)
    ax.imshow(img.numpy())
    ax.set_title("label = %d"%label)
    ax.set_xticks([])
    ax.set_yticks([])
plt.show()
```

2，定义模型

```
class CnnModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = nn.ModuleList([
            nn.Conv2d(in_channels=1,out_channels=32,kernel_size = 3),
            nn.MaxPool2d(kernel_size = 2,stride = 2),
            nn.Conv2d(in_channels=32,out_channels=64,kernel_size = 5),
            nn.MaxPool2d(kernel_size = 2,stride = 2),
            nn.Dropout2d(p = 0.1),
            nn.AdaptiveMaxPool2d((1,1)),
            nn.Flatten(),
            nn.Linear(64,32),
            nn.ReLU(),
            nn.Linear(32,10)]
        )
    def forward(self,x):
        for layer in self.layers:
            x = layer(x)
        return x

net = CnnModel()
model = torchkeras.Model(net)
model.summary(input_shape=(1,32,32))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 32, 30, 30]             320
         MaxPool2d-2          [-1, 32, 15, 15]               0
            Conv2d-3          [-1, 64, 11, 11]          51,264
         MaxPool2d-4            [-1, 64, 5, 5]               0
         Dropout2d-5            [-1, 64, 5, 5]               0
 AdaptiveMaxPool2d-6            [-1, 64, 1, 1]               0
           Flatten-7                  [-1, 64]               0
            Linear-8                  [-1, 32]           2,080
              ReLU-9                  [-1, 32]               0
           Linear-10                  [-1, 10]             330
================================================================
Total params: 53,994
Trainable params: 53,994
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.003906
Forward/backward pass size (MB): 0.359695
Params size (MB): 0.205971
Estimated Total Size (MB): 0.569572
----------------------------------------------------------------
```

3，训练模型

```
from sklearn.metrics import accuracy_score

def accuracy(y_pred,y_true):
    y_pred_cls = torch.argmax(nn.Softmax(dim=1)(y_pred),dim=1).data
    return accuracy_score(y_true.cpu().numpy(),y_pred_cls.cpu().numpy())
    # 注意此处要将数据先移动到cpu上，然后才能转换成numpy数组

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

model.compile(loss_func = nn.CrossEntropyLoss(),
              optimizer= torch.optim.Adam(model.parameters(),lr = 0.02),
              metrics_dict={"accuracy":accuracy},device = device) # 注意此处compile时指定了device

dfhistory = model.fit(3,dl_train = dl_train, dl_val=dl_valid, log_step_freq=100)
```

```
Start Training ...

================================================================================2020-06-27 00:24:29
{'step': 100, 'loss': 1.063, 'accuracy': 0.619}
{'step': 200, 'loss': 0.681, 'accuracy': 0.764}
{'step': 300, 'loss': 0.534, 'accuracy': 0.818}
{'step': 400, 'loss': 0.458, 'accuracy': 0.847}

 +-------+-------+----------+----------+--------------+
| epoch |  loss | accuracy | val_loss | val_accuracy |
+-------+-------+----------+----------+--------------+
|   1   | 0.412 |  0.863   |  0.128   |    0.961     |
+-------+-------+----------+----------+--------------+

================================================================================2020-06-27 00:24:35
{'step': 100, 'loss': 0.147, 'accuracy': 0.956}
{'step': 200, 'loss': 0.156, 'accuracy': 0.954}
{'step': 300, 'loss': 0.156, 'accuracy': 0.954}
{'step': 400, 'loss': 0.157, 'accuracy': 0.955}

 +-------+-------+----------+----------+--------------+
| epoch |  loss | accuracy | val_loss | val_accuracy |
+-------+-------+----------+----------+--------------+
|   2   | 0.153 |  0.956   |  0.085   |    0.976     |
+-------+-------+----------+----------+--------------+

================================================================================2020-06-27 00:24:42
{'step': 100, 'loss': 0.126, 'accuracy': 0.965}
{'step': 200, 'loss': 0.147, 'accuracy': 0.96}
{'step': 300, 'loss': 0.153, 'accuracy': 0.959}
{'step': 400, 'loss': 0.147, 'accuracy': 0.96}

 +-------+-------+----------+----------+--------------+
| epoch |  loss | accuracy | val_loss | val_accuracy |
+-------+-------+----------+----------+--------------+
|   3   | 0.146 |   0.96   |  0.119   |    0.968     |
+-------+-------+----------+----------+--------------+

================================================================================2020-06-27 00:24:48
Finished Training...
```

4，评估模型

```
%matplotlib inline
%config InlineBackend.figure_format = 'svg'

import matplotlib.pyplot as plt

def plot_metric(dfhistory, metric):
    train_metrics = dfhistory[metric]
    val_metrics = dfhistory['val_'+metric]
    epochs = range(1, len(train_metrics) + 1)
    plt.plot(epochs, train_metrics, 'bo--')
    plt.plot(epochs, val_metrics, 'ro-')
    plt.title('Training and validation '+ metric)
    plt.xlabel("Epochs")
    plt.ylabel(metric)
    plt.legend(["train_"+metric, 'val_'+metric])
    plt.show()


plot_metric(dfhistory,"loss")


plot_metric(dfhistory,"accuracy")


model.evaluate(dl_valid)


{'val_accuracy': 0.967068829113924, 'val_loss': 0.11601964030650598}
```

5，使用模型

```
model.predict(dl_valid)[0:10]
```

```
tensor([[ -9.2092,   3.1997,   1.4028,  -2.7135,  -0.7320,  -2.0518, -20.4938,
          14.6774,   1.7616,   5.8549],
        [  2.8509,   4.9781,  18.0946,   0.0928,  -1.6061,  -4.1437,   4.8697,
           3.8811,   4.3869,  -3.5929],
        [-22.5231,  13.6643,   5.0244, -11.0188, -16.8147,  -9.5894,  -6.2556,
         -10.5648, -12.1022, -19.4685],
        [ 23.2670, -12.0711,  -7.3968,  -8.2715,  -1.0915, -12.6050,   8.0444,
         -16.9339,   1.8827,  -0.2497],
        [ -4.1159,   3.2102,   0.4971, -11.8064,  12.1460,  -5.1650,  -6.5918,
           1.0088,   0.8362,   2.5132],
        [-26.1764,  15.6251,   6.1191, -12.2424, -13.9725, -10.0540,  -7.8669,
          -5.9602, -11.1944, -18.7890],
        [ -5.0602,   3.3779,  -0.6647,  -8.5185,  10.0320,  -5.5107,  -6.9579,
           2.3811,   0.2542,   3.2860],
        [  4.1017,  -0.4282,   7.2220,   3.3700,  -3.6813,   1.1576,  -1.8479,
           0.7450,   3.9768,   6.2640],
        [  1.9689,  -0.3960,   7.4414, -10.4789,   2.7066,   1.7482,   5.7971,
          -4.5808,   3.0911,  -5.1971],
        [ -2.9680,  -1.2369,  -0.0829,  -1.8577,   1.9380,  -0.8374,  -8.2207,
           3.5060,   3.8735,  13.6762]], device='cuda:0')
```

6，保存模型

```
# save the model parameters
torch.save(model.state_dict(), "model_parameter.pkl")

model_clone = torchkeras.Model(CnnModel())
model_clone.load_state_dict(torch.load("model_parameter.pkl"))

model_clone.compile(loss_func = nn.CrossEntropyLoss(),
             optimizer= torch.optim.Adam(model.parameters(),lr = 0.02),
             metrics_dict={"accuracy":accuracy},device = device) # 注意此处compile时指定了device

model_clone.evaluate(dl_valid)
```

```
{'val_accuracy': 0.967068829113924, 'val_loss': 0.11601964030650598}
```

四，torchkeras使用多GPU范例

注：以下范例需要在有多个GPU的机器上跑。如果在单GPU的机器上跑，也能跑通，但是实际上使用的是单个GPU。

1，准备数据

```
import torch
from torch import nn

import torchvision
from torchvision import transforms

import torchkeras


transform = transforms.Compose([transforms.ToTensor()])

ds_train = torchvision.datasets.MNIST(root="./data/minist/",train=True,download=True,transform=t
ds_valid = torchvision.datasets.MNIST(root="./data/minist/",train=False,download=True,transform=

dl_train =  torch.utils.data.DataLoader(ds_train, batch_size=128, shuffle=True, num_workers=4)
dl_valid =  torch.utils.data.DataLoader(ds_valid, batch_size=128, shuffle=False, num_workers=4)

print(len(ds_train))
print(len(ds_valid))
```

2，定义模型

```
class CnnModule(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = nn.ModuleList([
            nn.Conv2d(in_channels=1,out_channels=32,kernel_size = 3),
            nn.MaxPool2d(kernel_size = 2,stride = 2),
            nn.Conv2d(in_channels=32,out_channels=64,kernel_size = 5),
            nn.MaxPool2d(kernel_size = 2,stride = 2),
            nn.Dropout2d(p = 0.1),
            nn.AdaptiveMaxPool2d((1,1)),
            nn.Flatten(),
            nn.Linear(64,32),
            nn.ReLU(),
            nn.Linear(32,10)]
        )
    def forward(self,x):
        for layer in self.layers:
            x = layer(x)
        return x

net = nn.DataParallel(CnnModule())  #Attention this line!!!
model = torchkeras.Model(net)

model.summary(input_shape=(1,32,32))
```

3，训练模型

```
from sklearn.metrics import accuracy_score

def accuracy(y_pred,y_true):
    y_pred_cls = torch.argmax(nn.Softmax(dim=1)(y_pred),dim=1).data
    return accuracy_score(y_true.cpu().numpy(),y_pred_cls.cpu().numpy())
    # 注意此处要将数据先移动到cpu上，然后才能转换成numpy数组

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model.compile(loss_func = nn.CrossEntropyLoss(),
              optimizer= torch.optim.Adam(model.parameters(),lr = 0.02),
              metrics_dict={"accuracy":accuracy},device = device) # 注意此处compile时指定了device

dfhistory = model.fit(3,dl_train = dl_train, dl_val=dl_valid, log_step_freq=100)
```

```
Start Training ...

================================================================================2020-06-27 00:24:29
{'step': 100, 'loss': 1.063, 'accuracy': 0.619}
{'step': 200, 'loss': 0.681, 'accuracy': 0.764}
{'step': 300, 'loss': 0.534, 'accuracy': 0.818}
{'step': 400, 'loss': 0.458, 'accuracy': 0.847}

 +-------+-------+----------+----------+--------------+
| epoch |  loss | accuracy | val_loss | val_accuracy |
+-------+-------+----------+----------+--------------+
|   1   | 0.412 |  0.863   |  0.128   |    0.961     |
+-------+-------+----------+----------+--------------+

================================================================================2020-06-27 00:24:35
{'step': 100, 'loss': 0.147, 'accuracy': 0.956}
{'step': 200, 'loss': 0.156, 'accuracy': 0.954}
{'step': 300, 'loss': 0.156, 'accuracy': 0.954}
{'step': 400, 'loss': 0.157, 'accuracy': 0.955}

 +-------+-------+----------+----------+--------------+
| epoch |  loss | accuracy | val_loss | val_accuracy |
+-------+-------+----------+----------+--------------+
|   2   | 0.153 |  0.956   |  0.085   |    0.976     |
+-------+-------+----------+----------+--------------+

================================================================================2020-06-27 00:24:42
{'step': 100, 'loss': 0.126, 'accuracy': 0.965}
{'step': 200, 'loss': 0.147, 'accuracy': 0.96}
{'step': 300, 'loss': 0.153, 'accuracy': 0.959}
{'step': 400, 'loss': 0.147, 'accuracy': 0.96}

 +-------+-------+----------+----------+--------------+
| epoch |  loss | accuracy | val_loss | val_accuracy |
+-------+-------+----------+----------+--------------+
|   3   | 0.146 |   0.96   |  0.119   |    0.968     |
+-------+-------+----------+----------+--------------+

================================================================================2020-06-27 00:24:48
Finished Training...
```

4，评估模型

```
%matplotlib inline
%config InlineBackend.figure_format = 'svg'

import matplotlib.pyplot as plt

def plot_metric(dfhistory, metric):
    train_metrics = dfhistory[metric]
    val_metrics = dfhistory['val_'+metric]
    epochs = range(1, len(train_metrics) + 1)
    plt.plot(epochs, train_metrics, 'bo--')
    plt.plot(epochs, val_metrics, 'ro-')
    plt.title('Training and validation '+ metric)
    plt.xlabel("Epochs")
    plt.ylabel(metric)
    plt.legend(["train_"+metric, 'val_'+metric])
    plt.show()


plot_metric(dfhistory, "loss")


plot_metric(dfhistory,"accuracy")


model.evaluate(dl_valid)


{'val_accuracy': 0.9603441455696202, 'val_loss': 0.14203246376371081}
```

5，使用模型

```
model.predict(dl_valid)[0:10]
```

```
tensor([[ -9.2092,   3.1997,   1.4028,  -2.7135,  -0.7320,  -2.0518, -20.4938,
          14.6774,   1.7616,   5.8549],
        [  2.8509,   4.9781,  18.0946,   0.0928,  -1.6061,  -4.1437,   4.8697,
           3.8811,   4.3869,  -3.5929],
        [-22.5231,  13.6643,   5.0244, -11.0188, -16.8147,  -9.5894,  -6.2556,
         -10.5648, -12.1022, -19.4685],
        [ 23.2670, -12.0711,  -7.3968,  -8.2715,  -1.0915, -12.6050,   8.0444,
         -16.9339,   1.8827,  -0.2497],
        [ -4.1159,   3.2102,   0.4971, -11.8064,  12.1460,  -5.1650,  -6.5918,
           1.0088,   0.8362,   2.5132],
        [-26.1764,  15.6251,   6.1191, -12.2424, -13.9725, -10.0540,  -7.8669,
          -5.9602, -11.1944, -18.7890],
        [ -5.0602,   3.3779,  -0.6647,  -8.5185,  10.0320,  -5.5107,  -6.9579,
           2.3811,   0.2542,   3.2860],
        [  4.1017,  -0.4282,   7.2220,   3.3700,  -3.6813,   1.1576,  -1.8479,
           0.7450,   3.9768,   6.2640],
        [  1.9689,  -0.3960,   7.4414, -10.4789,   2.7066,   1.7482,   5.7971,
          -4.5808,   3.0911,  -5.1971],
        [ -2.9680,  -1.2369,  -0.0829,  -1.8577,   1.9380,  -0.8374,  -8.2207,
           3.5060,   3.8735,  13.6762]], device='cuda:0')
```

6，保存模型

```
# save the model parameters
torch.save(model.net.module.state_dict(), "model_parameter.pkl")

net_clone = CnnModel()
net_clone.load_state_dict(torch.load("model_parameter.pkl"))

model_clone = torchkeras.Model(net_clone)
model_clone.compile(loss_func = nn.CrossEntropyLoss(),
             optimizer= torch.optim.Adam(model.parameters(),lr = 0.02),
             metrics_dict={"accuracy":accuracy},device = device)
model_clone.evaluate(dl_valid)
```

```
{'val_accuracy': 0.9603441455696202, 'val_loss': 0.14203246376371081}
```