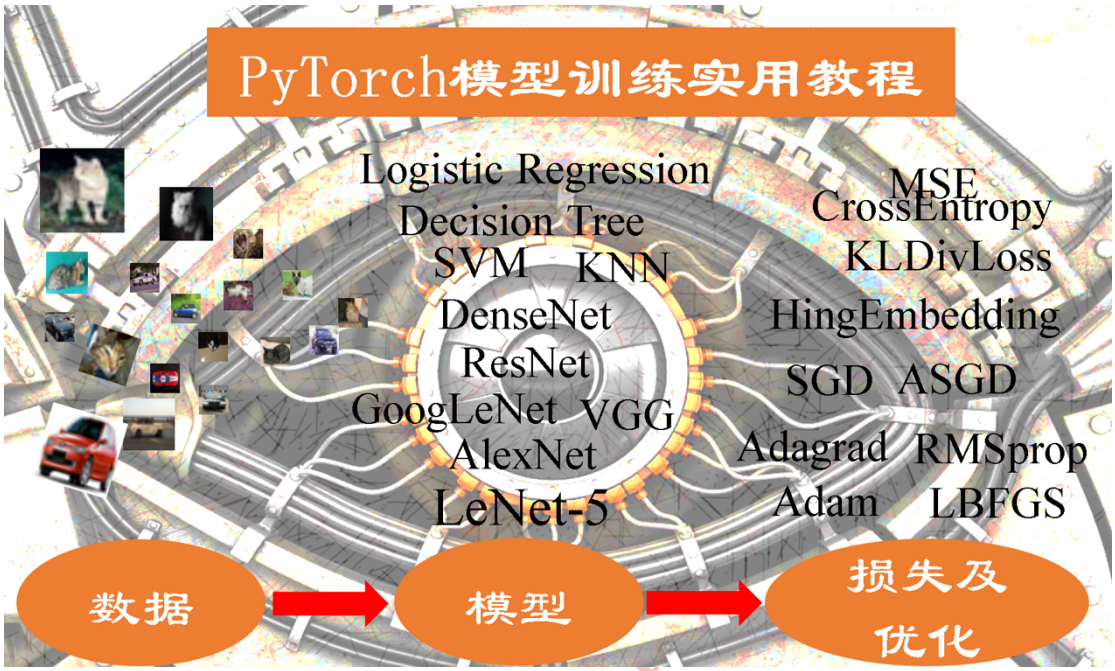
PyTorch 模型训练实用教程



前言：

自 2017 年 1 月 PyTorch 推出以来，其热度持续上升，一度有赶超 TensorFlow 的趋势。PyTorch 能在短时间内被众多研究人员和工程师接受并推崇是因为其有着诸多优点，如采用 Python 语言、动态图机制、网络构建灵活以及拥有强大的社群等。因此，走上学习 PyTorch 的道路已刻不容缓。

本教程以实际应用、工程开发为目的，着重介绍模型训练过程中遇到的实际问题和方法。如上图所示，在机器学习模型开发中，主要涉及三大部分，分别是数据、模型和损失函数及优化器。本文也按顺序的依次介绍数据、模型和损失函数及优化器，从而给大家带来清晰的机器学习结构。

通过本教程，希望能够给大家带来一个清晰的模型训练结构。当模型训练遇到问题时，需要通过可视化工具对数据、模型、损失等内容进行观察，分析并定位问题出在数据部分？模型部分？还是优化器？只有这样不断的通过可视化诊断你的模型，不断的对症下药，才能训练出一个较满意的模型。

为什么写此教程：

前几年一直在用 Caffe 和 MatConvNet，近期转 PyTorch。当时只想快速地用上 PyTorch 进行模型开发，然而搜了一圈 PyTorch 的教程，并没有找到一款

适合的。很多 PyTorch 教程是从学习机器学习(深度学习)的角度出发，以 PyTorch 为工具进行编写，里面介绍很多模型，并且附上模型的 demo。

然而，工程应用开发中所遇到的问题并不是跑一个模型的 demo 就可以的，模型开发需要对数据的预处理、数据增强、模型定义、权值初始化、模型 Finetune、学习率调整策略、损失函数选取、优化器选取、可视化等等。鉴于此，我只能自己对着官方文档，一步一步地学习。

起初，只是做了一些学习笔记，后来觉得这些内容应该对大家有些许帮助，毕竟在互联网上很难找到这类内容的分享，于是此教程就诞生了。

本教程内容及结构：

本教程内容主要为在 PyTorch 中训练一个模型所可能涉及到的方法及函数，并且对 PyTorch 提供的**数据增强方法（22 个）、权值初始化方法（10 个）、损失函数（17 个）、优化器（6 个）及 tensorboardX 的方法（13 个）**进行了详细介绍。

本教程分为四章，结构与机器学习三大部分一致。

第一章，介绍数据的划分，预处理，数据增强；

第二章，介绍模型的定义，权值初始化，模型 Finetune；

第三章，介绍各种损失函数及优化器；

第四章，介绍可视化工具，用于监控数据、模型权及损失函数的变化。

本教程适用读者：

1. 想熟悉 PyTorch 使用的朋友；
2. 想采用 PyTorch 进行模型训练的朋友；
3. 正采用 PyTorch，但无有效机制去诊断模型的朋友；

干货直达：

项目代码：https://github.com/tensor-yu/PyTorch_Tutorial

意见反馈：yts3221@126.com

学习交流 QQ 群： 为了更好的帮助大家学习和理解 PyTorch 以及机器学习相关知识，特建立一个 QQ 群，供大家交流，本文的最新修改也会同步到 QQ 群及 GitHub。QQ 群号：671103375

目录

第一章 数 据

1.1 Cifar10 转 png

俗话说得好，巧妇难为无米之炊，若没有数据，我们什么也做不了。在本教程中，为了统一大家的数据，这里采用 cifar-10 的测试集，共 10000 张图片作为源数据，模拟真实场景中的数据。

第一步：下载 cifar-10-python.tar.gz

下载 cifar-10-python.tar.gz，存放到 /Data 文件夹下，并且解压，获得文件夹/Data/cifar-10-batches-py/

下载方式：

1. 官网：http://www.cs.toronto.edu/~kriz/cifar.html
2. 百度云： https://pan.baidu.com/s/1NGV8g2iBAhHwjQZWTjGEbg 提取码: p3rh

第二步：运行 1_1_cifar10_to_png.py

运行代码：Code/1_data_prepare/1_1_cifar10_to_png.py

可在文件夹 Data/cifar-10-png/raw_test/下看到 0-9 个文件夹，对应 9 个类别。

脚本中未将训练集解压出来，这里只是为了实验，因此未使用过多的数据。这里仅将测试集中的 10000 张图片解压出来，作为原始图片，将从这 10000 张图片中划分出训练集(train)，验证集(valid)，测试集(test)。

运行完成，在 Data/cifar-10-png/raw_test 下将有 10 个文件夹，对应 10 个类别

.
└── raw_test
├── 0
├── 1
├── 2
├── 3

```
├── 4
├── 5
├── 6
├── 7
├── 8
└── 9
```

接着进入下一步：划分训练集、验证集和测试集。

1.2 训练集、验证集和测试集的划分

上一小节，把 cifar-10 的测试集转换成了 png 图片，充当实验的原始数据。本小节，将把原始数据按 8:1:1 的比例划分为训练集(train set)、验证集(valid/dev set)和测试集(test set)。关于训练集、验证集和测试集的作用，可阅读博客：
https://blog.csdn.net/u011995719/article/details/77451213

运行 Code/1_data_prepare/1_2_split_dataset.py，将会获得以下三个文件夹/Data/train/ /Data/valid/ /Data/test/

数据划分完毕，下一步是制作存放有图片路径及其标签的 txt，PyTorch 依据该 txt 上的信息进行寻找图片，并读取图片数据和标签数据。

1.3 让 PyTorch 能读你的数据集

上一小节中，将源数据(10000 张图片)划分为训练集、验证集和测试集，接下来就要让 PyTorch 能读取这批数据。想让 PyTorch 能读取我们自己的数据，首先要了解 pytroch 读取图片的机制和流程，然后按流程编写代码。

Dataset 类

PyTorch 读取图片，主要是通过 Dataset 类，所以先简单了解一下 Dataset 类。Dataset 类作为所有的 datasets 的基类存在，所有的 datasets 都需要继承它，类似于 C++中的虚基类。

源码如下：

```
class Dataset(object):
"""An abstract class representing a Dataset.
All other datasets should subclass it. All subclasses should override
``__len__``, that provides the size of the dataset, and ``__getitem__``,
supporting integer indexing in range from 0 to len(self) exclusive.
"""
def __getitem__(self, index):
raise NotImplementedError
def __len__(self):
raise NotImplementedError
def __add__(self, other):
return ConcatDataset([self, other])
```

这里重点看 getitem 函数，getitem 接收一个 index，然后返回图片数据和标签，这个 index 通常指的是一个 list 的 index，这个 list 的每个元素就包含了图片数据的路径和标签信息。

然而，如何制作这个 list 呢，通常的方法是将图片的路径和标签信息存储在一个 txt 中，然后从该 txt 中读取。

那么读取自己数据的基本流程就是：

1. 制作存储了图片的路径和标签信息的 txt
2. 将这些信息转化为 list，该 list 每一个元素对应一个样本
3. 通过 getitem 函数，读取数据和标签，并返回数据和标签

在训练代码里是感觉不到这些操作的，只会看到通过 DataLoader 就可以获取一个 batch 的数据，其实触发去读取图片这些操作的是 DataLoader 里的__iter__(self)，后面会详细讲解读取过程。在本小节，主要讲 Dataset 子类。

因此，要让 PyTorch 能读取自己的数据集，只需要两步：

1. 制作图片数据的索引
2. 构建 Dataset 子类

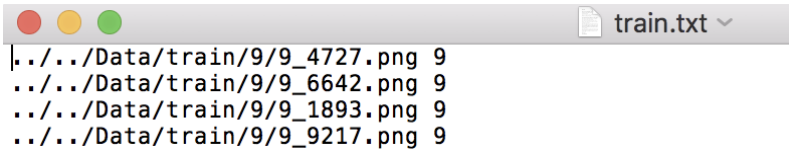
1. 制作图片数据的索引

这个比较简单，就是读取图片路径，标签，保存到 txt 文件中，这里注意格式就好

特别注意的是，txt 中的路径，是以训练时的那个 py 文件所在的目录为工作目录，所以这里需要提前算好相对路径！

运行代码 Code/1_data_prepare/1_3_generate_txt.py，即会在/Data/文件夹下面看到 train.txt valid.txt

txt 中是这样的：



2. 构建 Dataset 子类

下面是本实验构建的 Dataset 子类——MyDataset 类：

```
# coding: utf-8
from PIL import Image
from torch.utils.data import Dataset
class MyDataset(Dataset):
def __init__(self, txt_path, transform = None, target_transform = None):
fh = open(txt_path, 'r')
imgs = []
for line in fh:
line = line.rstrip()
words = line.split()
imgs.append((words[0], int(words[1])))
self.imgs = imgs
self.transform = transform
self.target_transform = target_transform
def __getitem__(self, index):
fn, label = self.imgs[index]
img = Image.open(fn).convert('RGB')
if self.transform is not None:
img = self.transform(img)
return img, label
def __len__(self):
return len(self.imgs)
```

首先看看初始化，初始化中从我们准备好的 txt 里获取图片的路径和标签，并且存储在 self.imgs，self.imgs 就是上面提到的 list，其一个元素对应一个样本的路径和标签，其实就是 txt 中的一行。

初始化中还会初始化 transform，transform 是一个 Compose 类型，里边有一个 list，list 中就会定义了各种对图像进行处理的操作，可以设置减均值，除标准差，随机裁剪，旋转，翻转，仿射变换等操作。

在这里我们可以知道，一张图片读取进来之后，会经过数据处理（数据增强），最终变成输入模型的数据。这里就有一点需要注意，PyTorch 的数据增强是将原始图片进行了处理，并不会生成新的一份图片，而是“覆盖”原图，当采用 randomcrop 之类的随机操作时，每个 epoch 输入进来的图片几乎不会是一模一样的，这达到了样本多样性的功能。

然后看看核心的 getitem 函数：

第一行：self.imgs 是一个 list，也就是一开始提到的 list，self.imgs 的一个元素是一个 str，包含图片路径，图片标签，这些信息是从 txt 文件中读取

第二行：利用 Image.open 对图片进行读取，img 类型为 Image ，mode='RGB'

第三行与第四行： 对图片进行处理，这个 transform 里边可以实现 减均值，除标准差，随机裁剪，旋转，翻转，放射变换，等等操作，这个放在后面会详细讲解。

当 Mydataset 构建好，剩下的操作就交给 DataLoder，在 DataLoder 中，会触发 Mydataset 中的 getiterm 函数读取一张图片的数据和标签，并拼接成一个 batch 返回，作为模型真正的输入。下一小节将会通过一个小例子，介绍 DataLoder 是如何获取一个 batch，以及一张图片是如何被 PyTorch 读取，最终变为模型的输入的。

1.4 图片从硬盘到模型

上小节中介绍了如何构建自己的 Dataset 子类——MyDataset，在 MyDataset 中，主要获取图片的索引以及定义如何通过索引读取图片及其标签。但是要触发 MyDataset 去读取图片及其标签却是在数据加载器 DataLoder 中。本小节，将进行单步调试，学习图片是如何从硬盘上流到模型的输入口的，并观察图片经历了哪些处理。

对应代码：

/Code/main_training/main.py

大体流程:

1. main.py: train_data = MyDataset(txt_path=train_txt_path, ...） --->

2. main.py: train_loader = DataLoader(dataset=train_data, ...) --->

3. main.py: for i, data in enumerate(train_loader, 0) --->

4. dataloder.py: class DataLoader(): def __iter__(self): return _DataLoaderIter(self) --->

5. dataloder.py: class _DataLoderIter(): def __next__(self): batch = self.collate_fn([self.dataset[i] for i in indices]) --->

6. tool.py: class MyDataset(): def __getitem__(): img = Image.open(fn).convert('RGB') --->

7. tool.py: class MyDataset(): img = self.transform(img) --->

8. main.py: inputs, labels = data inputs, labels = Variable(inputs), Variable(labels) outputs = net(inputs)

一句话概括就是，从 MyDataset 来，到 MyDataset 去。

一开始通过 MyDataset 创建一个实例，在该实例中有路径，有读取图片的方法(函数)。

然后需要 pytroch 的一系列规范化流程，在第 6 步中，才会调用 MyDataset 中的__getitem__()函数，最终通过 Image.open()读取图片数据。

然后对原始图片数据进行一系列预处理(transform 中设置)，最后回到 main.py，对数据进行转换成 Variable 类型，最终成为模型的输入。

流程详细描述：

1. 从 MyDataset 类中初始化 txt，txt 中有图片路径和标签

2. 初始化 DataLoder 时，将 train_data 传入，从而使 DataLoder 拥有图片的路径

3. 在一个 iteration 进行时，才读取一个 batch 的图片数据 enumerate()函数会返回可迭代数据的一个“元素”

在这里 data 是一个 batch 的图片数据和标签，data 是一个 list 喔

4. class DataLoader()中再调用 class _DataLoderIter()

5. 在 _DataLoderiter()类中会跳到__next__(self)函数，在该函数中会通过

indices = next(self.sample_iter) 获取一个 batch 的 indices

再通过

batch = self.collate_fn([self.dataset[i] for i in indices])获取一个 batch 的数据

在 batch = self.collate_fn([self.dataset[i] for i in indices])中会调用 self.collate_fn 函数

6. self.collate_fn 中会调用 MyDataset 类中的__getitem__()函数，在__getitem__()中通过 Image.open(fn).convert('RGB')读取图片

7. 通过 Image.open(fn).convert('RGB')读取图片之后，会对图片进行预处理，例如减均值，除以标准差，随机裁剪等等一系列提前设置好的操作。

具体 transform 的用法将用单独一小节介绍，最后返回 img，label，再通过 self.collate_fn 来拼接成一个 batch。一个 batch 是一个 list，有两个元素，第一个元素是图片数据，是一个 4D 的 Tensor，shape 为(64,3,32,32)，第二个元素是标签 shape 为(64)。

8. 将图片数据转换成 Variable 类型，然后称为模型真正的输入

inputs, labels = Variable(inputs), Variable(labels)

outputs = net(inputs)

通过了解图片从硬盘到模型的过程，我们可以更好的对数据做处理(减均值，除以标准差，裁剪，翻转，放射变换等等)，也可以灵活的为模型准备数据，最后总结两个需要注意的地方。

1. 图片是通过 Image.open()函数读取进来的，当涉及如下问题：

图片的通道顺序(RGB ? BGR ?)

图片是 w*h*c ？ c*w*h ？

像素值范围[0-1] or [0-255] ？

就要查看 MyDataset()类中 __getitem__() 下读取图片用的是什么方法

2. 从 MyDataset()类中 __getitem__() 函数中发现，PyTorch 做数据增强的方法是在原始图片上进行的，并覆盖原始图片，这一点需要注意。

1.5 数据增强与数据标准化

在实际应用过程中，我们会在数据进入模型之前进行一些预处理，例如数据中心化(仅减均值)，数据标准化(减均值，再除以标准差)，随机裁剪，旋转一定角度，镜像等一系列操作。PyTorch 有一系列数据增强方法供大家使用，下面将介绍这些方法。

在 PyTorch 中，这些数据增强方法放在了 transforms.py 文件中。这些数据处理可以满足我们大部分的需求，通过熟悉 transforms.py，以及 1.4 节中的内容，我们也可以自定义数据处理函数，实现自己的数据增强。

在本小节，从宏观地介绍 transform 的使用，在下一小节，将会详细介绍 transform 的所有操作。

transform 的使用

请查看/Code/main_trainingmain.py 中代码：

```
normMean = [0.4948052, 0.48568845, 0.44682974]
normStd = [0.24580306, 0.24236229, 0.2603115]
normTransform = transforms.Normalize(normMean, normStd)
trainTransform = transforms.Compose([
transforms.Resize(32),
transforms.RandomCrop(32, padding=4),
transforms.ToTensor(),
normTransform
])
validTransform = transforms.Compose([
transforms.ToTensor(),
normTransform
])
```

前三行设置均值，标准差，以及数据标准化：transforms.Normalize()函数，这里是以通道为单位进行计算均值，标准差。

然后用 transforms.Compose 将所需要进行的处理给 compose 起来，并且需要注意顺序！

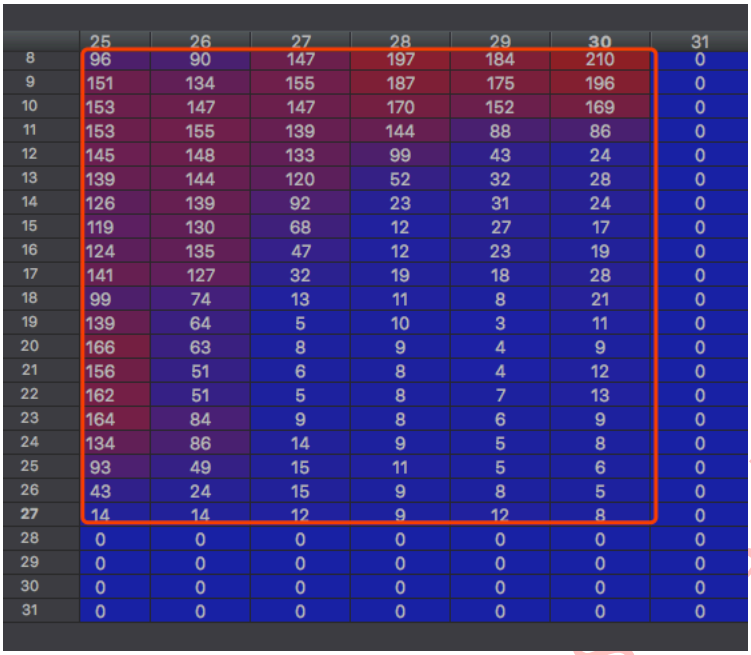
在训练时，依次对图片进行以下操作：

1. 随机裁剪
2. Totensor
3. 数据标准化(减均值，除以标准差)

1. 随机裁剪

第一个处理是随机裁剪，在裁剪之前先对图片的上下左右均填充上 4 个 pixel，值为 0，即变成一个 36*36 的数据，然后再随机进行 32*32 的裁剪。

例如下图，是经过 transforms.RandomCrop(32, padding=4),之后的图片，其中红色框是原始图片数据，31 列是填充的 0，28-31 行也是填充的 0.

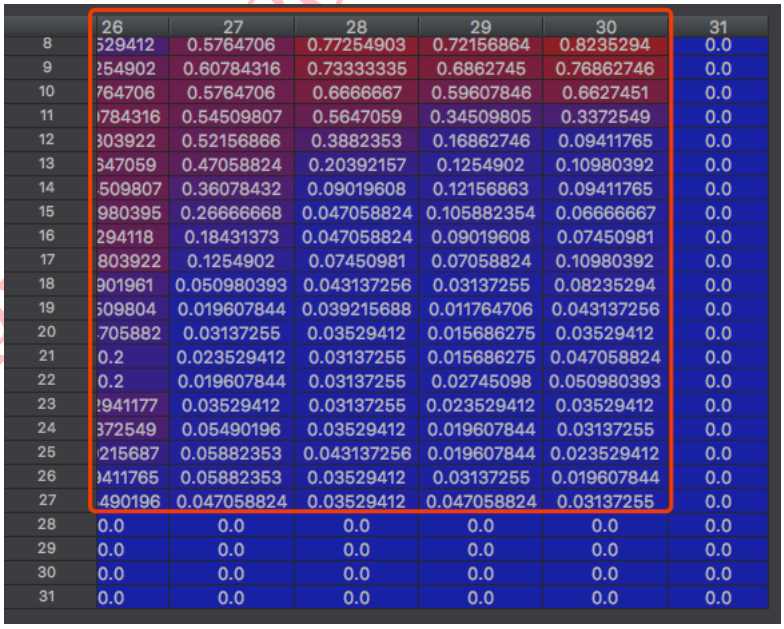


	25	26	27	28	29	30	31
8	96	90	147	197	184	210	0
9	151	134	155	187	175	196	0
10	153	147	147	170	152	169	0
11	153	155	139	144	88	86	0
12	145	148	133	99	43	24	0
13	139	144	120	52	32	28	0
14	126	139	92	23	31	24	0
15	119	130	68	12	27	17	0
16	124	135	47	12	23	19	0
17	141	127	32	19	18	28	0
18	99	74	13	11	8	21	0
19	139	64	5	10	3	11	0
20	166	63	8	9	4	9	0
21	156	51	6	8	4	12	0
22	162	51	5	8	7	13	0
23	164	84	9	8	6	9	0
24	134	86	14	9	5	8	0
25	93	49	15	11	5	6	0
26	43	24	15	9	8	5	0
27	14	14	12	9	12	8	0
28	0	0	0	0	0	0	0
29	0	0	0	0	0	0	0
30	0	0	0	0	0	0	0
31	0	0	0	0	0	0	0

2. Totensor

第二个处理是 transforms.ToTensor()

在这里会对数据进行 transpose，原来是 h*w*c，会经过 img = img.transpose(0, 1).transpose(0, 2).contiguous()，变成 c*h*w 再除以 255，使得像素值归一化至[0-1]之间，来看看 Red 通道。

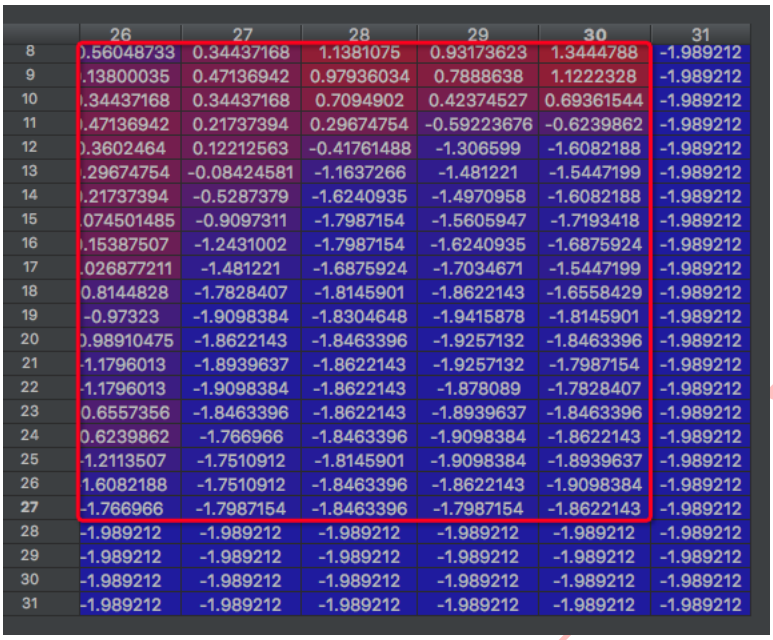


	26	27	28	29	30	31
8	529412	0.5764706	0.77254903	0.72156864	0.8235294	0.0
9	254902	0.60784316	0.73333335	0.6862745	0.76862746	0.0
10	764706	0.5764706	0.6666667	0.59607846	0.6627451	0.0
11	784316	0.54509807	0.5647059	0.34509805	0.3372549	0.0
12	303922	0.52156866	0.3882353	0.16862746	0.09411765	0.0
13	347059	0.47058824	0.20392157	0.1254902	0.10980392	0.0
14	509807	0.36078432	0.09019608	0.12156863	0.09411765	0.0
15	980395	0.26666668	0.047058824	0.105882354	0.06666667	0.0
16	294118	0.18431373	0.047058824	0.09019608	0.07450981	0.0
17	803922	0.1254902	0.07450981	0.07058824	0.10980392	0.0
18	901961	0.050980393	0.043137256	0.03137255	0.08235294	0.0
19	509804	0.019607844	0.039215688	0.011764706	0.043137256	0.0
20	705882	0.03137255	0.03529412	0.015686275	0.03529412	0.0
21	0.2	0.023529412	0.03137255	0.015686275	0.047058824	0.0
22	0.2	0.019607844	0.03137255	0.02745098	0.050980393	0.0
23	941177	0.03529412	0.03137255	0.023529412	0.03529412	0.0
24	372549	0.05490196	0.03529412	0.019607844	0.03137255	0.0
25	215687	0.05882353	0.043137256	0.019607844	0.023529412	0.0
26	9411765	0.05882353	0.03529412	0.03137255	0.019607844	0.0
27	490196	0.047058824	0.03529412	0.047058824	0.03137255	0.0
28	0.0	0.0	0.0	0.0	0.0	0.0
29	0.0	0.0	0.0	0.0	0.0	0.0
30	0.0	0.0	0.0	0.0	0.0	0.0
31	0.0	0.0	0.0	0.0	0.0	0.0

来看看 27 行，30 列 原来是 8 的， 经过 ToTensor 之后变成：8/255= 0.03137255

3. 数据标准化(减均值，除以标准差)

第三个处理是对图像进行标准化，通过标准化之后，再来看看 Red 通道的数据：



	26	27	28	29	30	31
8	0.56048733	0.34437168	1.1381075	0.93173623	1.3444788	-1.989212
9	.13800035	0.47136942	0.97936034	0.7888638	1.1222328	-1.989212
10	.34437168	0.34437168	0.7094902	0.42374527	0.69361544	-1.989212
11	.47136942	0.21737394	0.29674754	-0.59223676	-0.6239862	-1.989212
12	0.3602464	0.12212563	-0.41761488	-1.306599	-1.6082188	-1.989212
13	.29674754	-0.08424581	-1.1637266	-1.481221	-1.5447199	-1.989212
14	.21737394	-0.5287379	-1.6240935	-1.4970958	-1.6082188	-1.989212
15	074501485	-0.9097311	-1.7987154	-1.5605947	-1.7193418	-1.989212
16	.15387507	-1.2431002	-1.7987154	-1.6240935	-1.6875924	-1.989212
17	.026877211	-1.481221	-1.6875924	-1.7034671	-1.5447199	-1.989212
18	0.8144828	-1.7828407	-1.8145901	-1.8622143	-1.6558429	-1.989212
19	-0.97323	-1.9098384	-1.8304648	-1.9415878	-1.8145901	-1.989212
20	0.98910475	-1.8622143	-1.8463396	-1.9257132	-1.8463396	-1.989212
21	-1.1796013	-1.8939637	-1.8622143	-1.9257132	-1.7987154	-1.989212
22	-1.1796013	-1.9098384	-1.8622143	-1.878089	-1.7828407	-1.989212
23	0.6557356	-1.8463396	-1.8622143	-1.8939637	-1.8463396	-1.989212
24	0.6239862	-1.766966	-1.8463396	-1.9098384	-1.8622143	-1.989212
25	-1.2113507	-1.7510912	-1.8145901	-1.9098384	-1.8939637	-1.989212
26	1.6082188	-1.7510912	-1.8463396	-1.8622143	-1.9098384	-1.989212
27	-1.766966	-1.7987154	-1.8463396	-1.7987154	-1.8622143	-1.989212
28	-1.989212	-1.989212	-1.989212	-1.989212	-1.989212	-1.989212
29	-1.989212	-1.989212	-1.989212	-1.989212	-1.989212	-1.989212
30	-1.989212	-1.989212	-1.989212	-1.989212	-1.989212	-1.989212
31	-1.989212	-1.989212	-1.989212	-1.989212	-1.989212	-1.989212

至此，数据预处理完毕，最后转换成 Variable 类型，就是输入网络模型的数据了。

细心的朋友可能会发现，在进行 Normalize 时，需要设置均值和方差，在这里直接给出了，但在实际应用中是要去训练集中计算的，天下可没有免费的午餐。这里给出计算训练集的均值和方差的脚本：/Code/1_data_prepare/1_5_compute_mean.py

1.6 transforms 的二十二个方法

本小节对 transforms.py 中的各个预处理方法进行介绍和总结。主要从官方文档中总结而来，官方文档只是将方法陈列，没有归纳总结，顺序很乱，这里总结一共有四大类，方便大家索引：

1. 裁剪——Crop

中心裁剪：transforms.CenterCrop

随机裁剪：transforms.RandomCrop

随机长宽比裁剪：transforms.RandomResizedCrop

上下左右中心裁剪：transforms.FiveCrop

上下左右中心裁剪后翻转，transforms.TenCrop

2. 翻转和旋转——Flip and Rotation

依概率 p 水平翻转：transforms.RandomHorizontalFlip(p=0.5)

依概率 p 垂直翻转：transforms.RandomVerticalFlip(p=0.5)

随机旋转：transforms.RandomRotation

3.图像变换

resize：transforms.Resize

标准化：transforms.Normalize

转为 tensor，并归一化至[0-1]：transforms.ToTensor

填充：transforms.Pad

修改亮度、对比度和饱和度：transforms.ColorJitter

转灰度图：transforms.Grayscale

线性变换：transforms.LinearTransformation()

仿射变换：transforms.RandomAffine

依概率 p 转为灰度图：transforms.RandomGrayscale

将数据转换为 PILImage：transforms.ToPILImage

transforms.Lambda：Apply a user-defined lambda as a transform.

4.对 transforms 操作，使数据增强更灵活

transforms.RandomChoice(transforms)， 从给定的一系列 transforms 中选一个进行操作

transforms.RandomApply(transforms, p=0.5)，给一个 transform 加上概率，依概率进行操作

transforms.RandomOrder，将 transforms 中的操作随机打乱

一、 裁剪——Crop

1.随机裁剪：transforms.RandomCrop

class **torchvision.transforms.RandomCrop**(*size, padding=None, pad_if_needed=False, fill=0, padding_mode='constant'*)

功能：依据给定的 size 随机裁剪

参数：

size- (sequence or int)，若为 sequence,则为(h,w)，若为 int，则(size,size)

padding-(sequence or int, optional)，此参数是设置填充多少个 pixel。

当为 int 时，图像上下左右均填充 int 个，例如 padding=4，则上下左右均填充 4 个 pixel，若为 32*32，则会变成 40*40。

当为 sequence 时，若有 2 个数，则第一个数表示左右扩充多少，第二个数表示上下的。当有 4 个数时，则为左，上，右，下。

fill- (int or tuple) 填充的值是什么（仅当填充模式为 constant 时有用）。int 时，各通道均填充该值，当长度为 3 的 tuple 时，表示 RGB 通道需要填充的值。

padding_mode- 填充模式，这里提供了 4 种填充模式，1.constant，常量。2.edge 按照图片边缘的像素值来填充。3.reflect，暂不了解。 4. symmetric，暂不了解。

2.中心裁剪：transforms.CenterCrop

class **torchvision.transforms.CenterCrop(*size*)**

功能：依据给定的 size 从中心裁剪

参数：

size- (sequence or int)，若为 sequence,则为(h,w)，若为 int，则(size,size)

3.随机长宽比裁剪 transforms.RandomResizedCrop

class **torchvision.transforms.RandomResizedCrop(*size, scale=(0.08, 1.0), ratio=(0.75, 1.33333333333333), interpolation=2*)**

功能：随机大小，随机长宽比裁剪原始图片，最后将图片 resize 到设定好的 size

参数：

size- 输出的分辨率

scale- 随机 crop 的大小区间，如 scale=(0.08, 1.0)，表示随机 crop 出来的图片会在的 0.08 倍至 1 倍之间。

ratio- 随机长宽比设置

interpolation- 插值的方法，默认为双线性插值(PIL.Image.BILINEAR)

4.上下左右中心裁剪：transforms.FiveCrop

class **torchvision.transforms.FiveCrop(*size*)**

功能：对图片进行上下左右以及中心裁剪，获得 5 张图片，返回一个 4D-tensor

参数：

size- (sequence or int)，若为 sequence,则为(h,w)，若为 int，则(size,size)

5.上下左右中心裁剪后翻转: transforms.TenCrop

class **torchvision.transforms.TenCrop**(*size, vertical_flip=False*)

功能：对图片进行上下左右以及中心裁剪，然后全部翻转（水平或者垂直），获得 10 张图片，返回一个 4D-tensor。

参数：

size- (sequence or int)，若为 sequence,则为(h,w)，若为 int，则(size,size)

vertical_flip (*bool*) - 是否垂直翻转，默认为 flase，即默认为水平翻转

二、翻转和旋转——Flip and Rotation

6.依概率 p 水平翻转 transforms.RandomHorizontalFlip

class **torchvision.transforms.RandomHorizontalFlip**(*p=0.5*)

功能：依据概率 p 对 PIL 图片进行水平翻转

参数：

p- 概率，默认值为 0.5

7.依概率 p 垂直翻转 transforms.RandomVerticalFlip

class **torchvision.transforms.RandomVerticalFlip**(*p=0.5*)

功能：依据概率 p 对 PIL 图片进行垂直翻转

参数：

p- 概率，默认值为 0.5

8.随机旋转：transforms.RandomRotation

class **torchvision.transforms.RandomRotation**(*degrees, resample=False, expand=False, center=None*)

功能：依 degrees 随机旋转一定角度

参数：

degress- (*sequence or float or int*)，若为单个数，如 30，则表示在（-30，+30）之间随机旋转

若为 sequence，如(30，60)，则表示在 30-60 度之间随机旋转

resample- 重采样方法选择，可选

PIL.Image.NEAREST, PIL.Image.BILINEAR, PIL.Image.BICUBIC，默认为最近邻

expand- ?

center- 可选为中心旋转还是左上角旋转

三、图像变换

9.resize：transforms.Resize

class **torchvision.transforms.Resize(*size, interpolation=2*)**

功能：重置图像分辨率

参数：

size- If size is an int, if height > width, then image will be rescaled to (size * height / width, size)，**所以建议 size 设定为 h*w**

interpolation- 插值方法选择，默认为 *PIL.Image.BILINEAR*

10.标准化：transforms.Normalize

class **torchvision.transforms.Normalize(*mean, std*)**

功能：对数据按通道进行标准化，即先减均值，再除以标准差，注意是 h*w*c

11.转为 tensor：transforms.ToTensor

class **torchvision.transforms.ToTensor**

功能：将 PIL Image 或者 ndarray 转换为 tensor，并且归一化至[0-1]

注意事项：归一化至[0-1]是直接除以 255，若自己的 ndarray 数据尺度有变化，则需要自行修改。

12.填充：transforms.Pad

class **torchvision.transforms.Pad(*padding, fill=0, padding_mode='constant'*)**

功能：对图像进行填充

参数：

padding-(sequence or int, optional)，此参数是设置填充多少个 pixel。

当为 int 时，图像上下左右均填充 int 个，例如 padding=4，则上下左右均填充 4 个 pixel，若为 32*32，则会变成 40*40。

当为 sequence 时，若有 2 个数，则第一个数表示左右扩充多少，第二个数表示上下的。当有 4 个数时，则为左，上，右，下。

fill- (int or tuple) 填充的值是什么（仅当填充模式为 constant 时有用）。int 时，各通道均填充该值，当长度为 3 的 tuple 时，表示 RGB 通道需要填充的值。

padding_mode- 填充模式，这里提供了 4 种填充模式，1.constant，常量。2.edge 按照图片边缘的像素值来填充。3.reflect，？ 4. symmetric，？

13.修改亮度、对比度和饱和度：transforms.ColorJitter

class **torchvision.transforms.ColorJitter**(*brightness=0, contrast=0, saturation=0, hue=0*)

功能：修改修改亮度、对比度和饱和度

14.转灰度图：transforms.Grayscale

class **torchvision.transforms.Grayscale**(*num_output_channels=1*)

功能：将图片转换为灰度图

参数：

num_output_channels- (int) ，当为 1 时，正常的灰度图，当为 3 时， 3 channel with r == g == b

15.线性变换：transforms.LinearTransformation()

class **torchvision.transforms.LinearTransformation**(*transformation_matrix*)

功能：对矩阵做线性变化，可用于白化处理！ **whitening: zero-center the data, compute the data covariance matrix**

参数：

transformation_matrix (*Tensor*) – tensor [D x D], D = C x H x W

16.仿射变换：transforms.RandomAffine

class **torchvision.transforms.RandomAffine**(*degrees, translate=None, scale=None, shear=None, resample=False, fillcolor=0*)

功能：仿射变换

17.依概率 p 转为灰度图：transforms.RandomGrayscale

class **torchvision.transforms.RandomGrayscale(*p=0.1*)**

功能：依概率 p 将图片转换为灰度图，若通道数为 3，则 3 channel with r == g == b

18.将数据转换为 PILImage：transforms.ToPILImage

class **torchvision.transforms.ToPILImage(*mode=None*)**

功能：将 tensor 或者 ndarray 的数据转换为 PIL Image 类型数据

参数：

mode- 为 None 时，为 1 通道， mode=3 通道默认转换为 RGB，4 通道默认转换为 RGBA

19.transforms.Lambda

Apply a user-defined lambda as a transform.

暂不了解，待补充。

四、对 transforms 操作，使数据增强更灵活

PyTorch 不仅可设置对图片的操作，还可以对这些操作进行随机选择、组合

20.transforms.RandomChoice(transforms)

功能：从给定的一系列 transforms 中选一个进行操作，randomly picked from a list

21.transforms.RandomApply(transforms, p=0.5)

功能：给一个 transform 加上概率，以一定的概率执行该操作

22.transforms.RandomOrder

功能：将 transforms 中的操作顺序随机打乱

第二章 模 型

第二章介绍关于网络模型的一系列内容，包括模型的定义，模型参数初始化方法，模型的保存和加载，模型的 finetune(本质上还是模型权值初始化)，首先介绍模型的定义。

2.1 模型的搭建

2.1.1 模型定义的三要

首先，必须继承 nn.Module 这个类，要让 PyTorch 知道这个类是一个 Module。

其次，在__init__(self)中设置好需要的“组件"(如 conv、pooling、Linear、BatchNorm 等)。

最后，在 forward(self, x)中用定义好的“组件”进行组装，就像搭积木，把网络结构搭建出来，这样一个模型就定义好了。

接下来，请看代码，在/Code/main_training/main.py 中可以看到定义了一个类 class Net(nn.Module)，先看__init__(self)函数

```
def __init__(self):
super(Net, self).__init__()
self.conv1 = nn.Conv2d(3, 6, 5)
self.pool1 = nn.MaxPool2d(2, 2)
self.conv2 = nn.Conv2d(6, 16, 5)
self.pool2 = nn.MaxPool2d(2, 2)
self.fc1 = nn.Linear(16 * 5 * 5, 120)
self.fc2 = nn.Linear(120, 84)
self.fc3 = nn.Linear(84, 10)
```

第一行是初始化，往后定义了一系列组件，如由 Conv2d 构成的 conv1，有 MaxPool2d 构成的 poo11，这些操作均由 torch.nn 提供，torch.nn 中的操作可查看文档：https://PyTorch.org/docs/stable/nn.html#。

当这些组件定义好之后，就可以定义 forward()函数，用来搭建网络结构，请看代码：

```
def forward(self, x):
x = self.pool1(F.relu(self.conv1(x)))
x = self.pool2(F.relu(self.conv2(x)))
x = x.view(-1, 16 * 5 * 5)
x = F.relu(self.fc1(x))
x = F.relu(self.fc2(x))
x = self.fc3(x)
return x
```

x 为模型的输入，第一行表示，x 经过 conv1，然后经过激活函数 relu，再经过 pool1 操作；

第二行于第一行一样；第三行，表示将 x 进行 reshape，为了后面做为全连接层的输入；

第四，第五行的操作都一样，先经过全连接层 fc，然后经过 relu；

第六行，模型的最终输出是 fc3 输出。

至此，一个模型定义完毕，接着就可以在后面进行使用。

例如，实例化一个模型 net = Net()，然后把输入 inputs 扔进去，outputs = net(inputs)，就可以得到输出 outputs。

2.1.2 模型定义多说两句

上面只是介绍了模型定义的要素和过程，但是在工程应用中会碰到各种各样的网络模型，这时，我们就需要一些实用工具来帮助我们定义模型了。

这里以 Resnet34 为例介绍“复杂”模型的定义，这部分代码从 github：

https://github.com/yuanlairuci110/PyTorch-best-practice-master/blob/master/models/ResNet34.py

上获取。

```
class ResidualBlock(nn.Module):
'''
实现子 module: Residual Block
'''
def __init__(self, inchannel, outchannel, stride=1, shortcut=None):
super(ResidualBlock, self).__init__()
```

```
self.left = nn.Sequential(
nn.Conv2d(inchannel, outchannel, 3, stride, 1, bias=False),
nn.BatchNorm2d(outchannel),
nn.ReLU(inplace=True),
nn.Conv2d(outchannel, outchannel, 3, 1, 1, bias=False),
nn.BatchNorm2d(outchannel) )
self.right = shortcut

def forward(self, x):
out = self.left(x)
residual = x if self.right is None else self.right(x)
out += residual
return F.relu(out)

class ResNet34(BasicModule):
'''
实现主 module：ResNet34
ResNet34 包含多个 layer，每个 layer 又包含多个 Residual block
用子 module 来实现 Residual block，用_make_layer 函数来实现 layer
'''
def __init__(self, num_classes=2):
super(ResNet34, self).__init__()
self.model_name = 'resnet34'

# 前几层：图像转换
self.pre = nn.Sequential(
nn.Conv2d(3, 64, 7, 2, 3, bias=False),
nn.BatchNorm2d(64),
nn.ReLU(inplace=True),
nn.MaxPool2d(3, 2, 1))
# 重复的 layer，分别有 3，4，6，3 个 residual block
self.layer1 = self._make_layer( 64, 128, 3)
```

```
self.layer2 = self._make_layer( 128, 256, 4, stride=2)
self.layer3 = self._make_layer( 256, 512, 6, stride=2)
self.layer4 = self._make_layer( 512, 512, 3, stride=2)

#分类用的全连接
self.fc = nn.Linear(512, num_classes)
def _make_layer(self, inchannel, outchannel, block_num, stride=1):
'''
构建 layer,包含多个 residual block
'''
shortcut = nn.Sequential(
nn.Conv2d(inchannel,outchannel,1,stride, bias=False),
nn.BatchNorm2d(outchannel))
layers = []
layers.append(ResidualBlock(inchannel, outchannel, stride, shortcut))
for i in range(1, block_num):
layers.append(ResidualBlock(outchannel, outchannel))
return nn.Sequential(*layers)
def forward(self, x):
x = self.pre(x)
x = self.layer1(x)
x = self.layer2(x)
x = self.layer3(x)
x = self.layer4(x)

x = F.avg_pool2d(x, 7)
x = x.view(x.size(0), -1)
return self.fc(x)
```

还是从**三要素**出发看看是怎么定义 Resnet34 的。

首先，继承 nn.Module；

其次，看__init__()函数，在__init__()中，定义了这些组件，self.pre，self.layer1-4, self.fc ；

最后，看 forward()，分别用了在__init__()中定义的一系列组件，并且用了 torch.nn.functional.avg_pool2d 这个操作

至此，网络定义完成。

以为就完了？怎么可能，__init__()函数中的组件是怎么定义的，在__init__()中出现了 torch.nn.Sequential

组件定义还调用函数_make_layer()，其中也用到了 torch.nn.Sequential，其中还调用了 ResidualBlock(nn.Module)，在 ResidualBlock(nn.Module)中有一次调用了 torch.nn.Sequential。

torch.nn.Sequential 到底是什么呢？为什么都在用呢？

2.1.3 nn.Sequetial

torch.nn.Sequential 其实就是 Sequential 容器，该容器将一系列操作按先后顺序给包起来，方便重复使用，

例如 Resnet 中有很多重复的 block，就可以用 Sequential 容器把重复的地方包起来。

官方文档中给出两个使用例子：

```
# Example of using Sequential
model = nn.Sequential( nn.Conv2d(1,20,5), nn.ReLU(), nn.Conv2d(20,64,5), nn.ReLU() ) #
Example of using Sequential with OrderedDict model = nn.Sequential(OrderedDict([ ('conv1',
nn.Conv2d(1,20,5)), ('relu1', nn.ReLU()), ('conv2', nn.Conv2d(20,64,5)), ('relu2', nn.ReLU()) ]))
```

小结：

模型的定义就是先**继承**，再**构建组件**，最后**组装**。

其中基本组件可从 torch.nn 中获取，或者从 torch.nn.functional 中获取，同时为了方便重复使用组件，可以使用 Sequential 容器将一系列组件包起来，最后在 forward()函数中将这些组件组装成你的模型。

2.2 权值初始化的十种方法

上一小节介绍了模型定义的方法，模型定义完成后，通常我们还需要对权值进行初始化，才能开始训练。

初始化方法会直接影响到模型的收敛与否，在本小节，将介绍如何对模型进行初始化。

2.2.1 权值初始化流程

总共两步，

第一步，先设定什么层用什么初始化方法，初始化方法在 torch.nn.init 中给出；

第二步，实例化一个模型之后，执行该函数，即可完成初始化。

我们的重点是在第一步，请看 main.py 中第一步的代码：

```
# 定义权值初始化
def initialize_weights(self):
for m in self.modules():
if isinstance(m, nn.Conv2d):
torch.nn.init.xavier_normal(m.weight.data)
if m.bias is not None:
m.bias.data.zero_()
elif isinstance(m, nn.BatchNorm2d):
m.weight.data.fill_(1)
m.bias.data.zero_()
elif isinstance(m, nn.Linear):
torch.nn.init.normal(m.weight.data, 0, 0.01)
m.bias.data.zero_()
```

这段代码基本流程是这样，先从 self.modules()中遍历每一层，然后判断各层属于什么类型，例如，是否是 nn.Conv2d、nn.BatchNorm2d、nn.Linear 等，然后根据不同类型的层，设定不同的权值初始化方法，例如，Xavier，kaiming，normal_，uniform_等。

Ps: kaiming 也称之为 MSRA 初始化，当年何恺明还在微软亚洲研究院，因而得名。

来看看第一行代码中的 self.modules()，源码在 torch/nn/modules/module.py 中

```
def modules(self):
for name, module in self.named_modules():
yield module
```

功能是：Returns an iterator over all modules in the network. 能依次返回模型中的各层，例如：

接着，判断 m 的类型，属于什么类型，可以看到当前 m 属于 Conv2d 类型，则进行如下初始化：

```
torch.nn.init.xavier_normal(m.weight.data)
if m.bias is not None:
m.bias.data.zero_()
```

以上代码表示采用 torch.nn.init.xavier_normal 方法对该层的 weight 进行初始化，并判断是否存在偏置(bias)，若存在，将 bias 初始化为全 0。

这样，该层就初始化完毕，参照以上流程，不断遍历模型的每一层，最终完成模型的初始化。

2.2.2 常用初始化方法

PyTorch 在 **torch.nn.init** 中提供了常用的初始化方法函数，这里简单介绍，方便查询使用。

介绍分两部分：

1. Xavier，kaiming 系列；
2. 其他方法分布

Xavier 初始化方法，论文在《Understanding the difficulty of training deep feedforward neural networks》

公式推导是从“方差一致性”出发，初始化的分布有均匀分布和正态分布两种。

1. Xavier 均匀分布

torch.nn.init.xavier_uniform_(tensor, gain=1)

xavier 初始化方法中服从均匀分布 U(−a,a) ， 分布的参数 a = gain * sqrt(6/fan_in+fan_out)，这里有一个 gain，增益的大小是依据激活函数类型来设定

eg：nn.init.xavier_uniform_(w, gain=nn.init.calculate_gain('relu'))

PS：上述初始化方法，也称为 Glorot initialization

2. Xavier 正态分布

torch.nn.init.xavier_normal_*(tensor, gain=1)*

xavier 初始化方法中服从正态分布，

mean=0,std = gain * sqrt(2/fan_in + fan_out)

kaiming 初始化方法，论文在《 Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification》，公式推导同样从“方差一致性”出法，kaiming 是针对 xavier 初始化方法在 relu 这一类激活函数表现不佳而提出的改进，详细可以参看论文。

3. kaiming 均匀分布

torch.nn.init.kaiming_uniform_*(tensor, a=0, mode='fan_in', nonlinearity='leaky_relu')*

此为均匀分布，U～（-bound, bound）, bound = sqrt(6/(1+a^2)*fan_in)

其中，a 为激活函数的负半轴的斜率，relu 是 0

mode- 可选为 fan_in 或 fan_out, fan_in 使正向传播时，方差一致; fan_out 使反向传播时，方差一致

nonlinearity- 可选 relu 和 leaky_relu ，默认值为 。 leaky_relu

nn.init.kaiming_uniform_(w, mode='fan_in', nonlinearity='relu')

4. kaiming 正态分布

torch.nn.init.kaiming_normal_*(tensor, a=0, mode='fan_in', nonlinearity='leaky_relu')*

此为 0 均值的正态分布，N～ (0,std)，其中 std = sqrt(2/(1+a^2)*fan_in)

其中，a 为激活函数的负半轴的斜率，relu 是 0

mode- 可选为 fan_in 或 fan_out, fan_in 使正向传播时，方差一致;fan_out 使反向传播时，方差一致

nonlinearity- 可选 relu 和 leaky_relu ，默认值为 。 leaky_relu

```
nn.init.kaiming_normal_(w, mode='fan_out', nonlinearity='relu')
```

2.其他

5. 均匀分布初始化

torch.nn.init.uniform_(*tensor, a=0, b=1*)

使值服从均匀分布 U(a,b)

6. 正态分布初始化

torch.nn.init.normal_(*tensor, mean=0, std=1*)

使值服从正态分布 N(mean, std)，默认值为 0，1

7. 常数初始化

torch.nn.init.constant_(*tensor, val*)

使值为常数 val `nn.init.constant_(w, 0.3)`

8. 单位矩阵初始化

torch.nn.init.eye_(*tensor*)

将二维 tensor 初始化为单位矩阵（the identity matrix）

9. 正交初始化

torch.nn.init.orthogonal_(*tensor, gain=1*)

使得 tensor 是正交的，论文:Exact solutions to the nonlinear dynamics of learning in deep linear neural networks" - Saxe, A. et al. (2013)

10. 稀疏初始化

torch.nn.init.sparse_(*tensor, sparsity, std=0.01*)

从正态分布 N～（0. std）中进行稀疏化，使每一个 column 有一部分为 0

sparsity- 每一个 column 稀疏的比例，即为 0 的比例

```
nn.init.sparse_(w, sparsity=0.1)
```

11. 计算增益

torch.nn.init.calculate_gain(*nonlinearity, param=None*)

nonlinearity	gain
Linear / Identity	1
Conv{1,2,3}D	1
Sigmoid	1
Tanh	5/3
ReLU	sqrt(2)
Leaky Relu	sqrt(2/1+neg_slop^2)

权值初始化杂谈

1. 从代码中发现，即使不进行初始化，我们模型的权值也不为空，而是有值的，这些值是在什么时候赋的呢？

其实，在创建网络实例的过程中，一旦调用 nn.Conv2d 的时候就会有对权值进行初始化

Conv2d 是继承_ConvNd，初始化赋值是在_ConvNd 当中的

self.weight = Parameter(torch.Tensor(

out_channels, in_channels // groups, *kernel_size))

这些值是创建一个 Tensor 时得到的，是一些很小的随机数。

2. 按需定义初始化方法，例如：

```
if isinstance(m, nn.Conv2d):
n = m.kernel_size[0] * m.kernel_size[1] * m.out_channels
m.weight.data.normal_(0, math.sqrt(2. / n))
```

2.3 模型 Finetune

上一小节，介绍了模型权值初始化，以及 PyTorch 自带的权值初始化方法函数。我们知道一个良好的权值初始化，可以使收敛速度加快，甚至可以获得更好的精度。而在实际应用中，我们通常采用一个已经训练模型的模型的权值参数作为我们模型的初始化参数，也称之为 Finetune，更宽泛的称之为迁移学习。迁移学习中的 Finetune 技术，本质上就是让我们新构建的模型，拥有一个较好的权值初始值。

finetune 权值初始化三步曲，finetune 就相当于给模型进行初始化，其流程共用三步：

第一步：保存模型，拥有一个预训练模型；

第二步：加载模型，把预训练模型中的权值取出来；

第三步：初始化，将权值对应的“放”到新模型中

一、Finetune 之权值初始化

在进行 finetune 之前我们需要拥有一个模型或者是模型参数，因此需要了解如何保存模型。官方文档中介绍了两种保存模型的方法，一种是保存整个模型，另外一种是仅保存模型参数（官方推荐用这种方法），这里采用官方推荐的方法。

第一步：保存模型参数

若拥有模型参数，可跳过这一步。

假设创建了一个 net = Net()，并且经过训练，通过以下方式保存：

```
torch.save(net.state_dict(), 'net_params.pkl')
```

第二步：加载模型

进行三步曲中的第二步，加载模型，这里只是加载模型的参数：

```
pretrained_dict = torch.load('net_params.pkl')
```

第三步：初始化

进行三步曲中的第三步，将取到的权值，对应的放到新模型中：

首先我们创建新模型，并且获取新模型的参数字典 net_state_dict：

```
net = Net() # 创建 net
net_state_dict = net.state_dict() # 获取已创建 net 的 state_dict
```

接着将 pretrained_dict 里不属于 net_state_dict 的键剔除掉：

```
pretrained_dict_1 = {k: v for k, v in pretrained_dict.items() if k in net_state_dict}
```

然后，用预训练模型的参数字典 对 新模型的参数字典 net_state_dict 进行更新：

```
net_state_dict.update(pretrained_dict_1)
```

最后，将更新了参数的字典 “放”回到网络中：

```
net.load_state_dict(net_state_dict)
```

这样，利用预训练模型参数对新模型的权值进行初始化过程就做完了。

采用 finetune 的训练过程中，有时候希望前面层的学习率低一些，改变不要太大，而后面的全连接层的学习率相对大一些。这时就需要对不同的层设置不同的学习率，下面就介绍如何为不同层配置不同的学习率。

二、不同层设置不同的学习率

在利用 pre-trained model 的参数做初始化之后，我们可能想让 fc 层更新相对快一些，而希望前面的权值更新小一些，这就可以通过为不同的层设置不同的学习率来达到此目的。

为不同层设置不同的学习率，主要通过优化器对多个参数组进行设置不同的参数。所以，只需要将原始的参数组，划分成两个，甚至更多的参数组，然后分别进行设置学习率。

这里将原始参数“切分”成 fc3 层参数和其余参数，为 fc3 层设置更大的学习率。

请看代码：

```
ignored_params = list(map(id, net.fc3.parameters())) # 返回的是 parameters 的 内存地址
base_params = filter(lambda p: id(p) not in ignored_params, net.parameters()) # 返回 base params 的 内存地址
optimizer = optim.SGD([
{'params': base_params},
{'params': net.fc3.parameters(), 'lr': 0.001*10}], 0.001, momentum=0.9, weight_decay=1e-4)
```

第一行+ 第二行的意思就是，将 fc3 层的参数 net.fc3.parameters()从原始参数 net.parameters()中剥离出来

base_params 就是剥离了 fc3 层的参数的其余参数，然后在优化器中为 fc3 层的参数单独设定学习率。

optimizer = optim.SGD(.....)这里的意思就是 base_params 中的层，用 0.001, momentum=0.9, weight_decay=1e-4

fc3 层设定学习率为： 0.001*10

完整代码位于 /Code/2_model/2_finetune.py

补充：

挑选出特定的层的机制是利用内存地址作为过滤条件，将需要单独设定的那部分参数，从总的参数中剔除。

base_params 是一个 list，每个元素是一个 Parameter 类

net.fc3.parameters() 是一个 <generator object parameters>

ignored_params = list(map(id, net.fc3.parameters()))

net.fc3.parameters() 是一个<generator object parameters at 0x11b63bf00>

所以迭代的返回其中的 parameter，这里有 weight 和 bias

最终返回 weight 和 bias 所在内存的地址

第三章 损失函数与优化器

通过前两章，我们准备好数据，设计好模型，接下来就是选择合适的损失函数，并且采用合适的优化器进行优化(训练)模型。本章中，将介绍 PyTorch 中的十七个损失函数，十个优化器和六个学习率调整方法。

3.1 PyTorch 的十七个损失函数

我们所说的优化，即优化网络权值使得损失函数值变小。但是，损失函数值变小是否能代表模型的分类/回归精度变高呢？那么多种损失函数，应该如何选择呢？请来了解 PyTorch 中给出的十七种损失函数吧。

请运行配套代码，代码中有详细解释，有手动计算，这些都有助于理解损失函数原理。

本小节配套代码： /Code/3_optimizer/3_1_lossFunction

1. L1loss

class **torch.nn.L1Loss**(*size_average=None, reduce=None*)

官方文档中仍有 reduction='elementwise_mean'参数，但代码实现中已经删除该参数

功能：

计算 output 和 target 之差的绝对值，可选返回同维度的 tensor 或者是一个标量。

计算公式：

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = |x_n - y_n|$$

参数：

reduce(bool)- 返回值是否为标量，默认为 True

size_average(bool)- 当 reduce=True 时有效。为 True 时，返回的 loss 为平均值；为 False 时，返回的各样本的 loss 之和。

实例：

/Code/3_optimizer/3_1_lossFunction/1_L1Loss.py

2. MSELoss

class **torch.nn.MSELoss**(*size_average=None, reduce=None, reduction='elementwise_mean'*)

官方文档中仍有 reduction='elementwise_mean'参数，但代码实现中已经删除该参数

功能：

计算 output 和 target 之差的平方，可选返回同维度的 tensor 或者是一个标量。

计算公式：

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = (x_n - y_n)^2$$

参数：

reduce(bool)- 返回值是否为标量，默认为 True

size_average(bool)- 当 reduce=True 时有效。为 True 时，返回的 loss 为平均值；为 False 时，返回的各样本的 loss 之和。

实例：

/Code/3_optimizer/3_1_lossFunction/2_MSELoss.py

3. CrossEntropyLoss

class **torch.nn.CrossEntropyLoss**(*weight=None, size_average=None, ignore_index=-100, reduce=None, reduction='elementwise_mean'*)

功能：

将输入经过 softmax 激活函数之后，再计算其与 target 的交叉熵损失。即该方法将 nn.LogSoftmax()和 nn.NLLLoss()进行了结合。严格意义上的交叉熵损失函数应该是 nn.NLLLoss()。

补充：小谈交叉熵损失函数

交叉熵损失(cross-entropy Loss) 又称为对数似然损失(Log-likelihood Loss)、对数损失；二分类时还可称之为逻辑斯谛回归损失(Logistic Loss)。交叉熵损失函数表达式为 L = - sigama(y_i * log(x_i))。pytroch 这里不是严格意义上的交叉熵损失函数，而是先将 input 经过 softmax 激活函数，将向量“归一化”成概率形式，然后再与 target 计算严格意义上交叉熵损失。

在多分类任务中，经常采用 softmax 激活函数+交叉熵损失函数，因为交叉熵描述了两个概率分布的差异，然而神经网络输出的是向量，并不是概率分布的形式。所以需要 softmax 激活函数将一个向量进行“归一化”成概率分布的形式，再采用交叉熵损失函数计算 loss。

再回顾 PyTorch 的 CrossEntropyLoss()，官方文档中提到时将 nn.LogSoftmax()和 nn.NLLLoss()进行了结合，nn.LogSoftmax() 相当于激活函数 ， nn.NLLLoss()是损失函数，将其结合，完整的是否可以叫做 softmax+交叉熵损失函数呢？

计算公式：

$$\text{loss}(x, class) = -\log\left(\frac{\exp(x[class])}{\sum_j \exp(x[j])}\right) = -x[class] + \log\left(\sum_j \exp(x[j])\right)$$

参数：

weight(Tensor)- 为每个类别的 loss 设置权值，常用于类别不均衡问题。weight 必须是 float 类型的 tensor，其长度要于类别 C 一致，即每一个类别都要设置有 weight。带 weight 的计算公式：

$$\text{loss}(x, class) = weight[class]\left(-x[class] + \log\left(\sum_j \exp(x[j])\right)\right)$$

size_average(bool)- 当 reduce=True 时有效。为 True 时，返回的 loss 为平均值；为 False 时，返回的各样本的 loss 之和。

reduce(bool)- 返回值是否为标量，默认为 True

ignore_index(int)- 忽略某一类别，不计算其 loss，其 loss 会为 0，并且，在采用 size_average 时，不会计算那一类的 loss，除的时候的分母也不会统计那一类的样本。

实例：

/Code/3_optimizer/3_1_lossFunction/3_CroosEntropyLoss.py

补充：

output 不仅可以是向量，还可以是图片，即对图像进行像素点的分类，这个例子可以从 NLLLoss()中看到，这在图像分割当中很有用。

4. NLLLoss

class **torch.nn.NLLLoss**(***weight=None, size_average=None, ignore_index=-100, reduce=None, reduction='elementwise_mean'***)

功能：

不好用言语描述其功能！请看计算公式：loss(input, class) = -input[class]。举个例，三分类任务，input=[-1.233, 2.657, 0.534]， 真实标签为 2（class=2），则 loss 为-0.534。就是对应类别上的输出，取一个负号！感觉被 NLLLoss 的名字欺骗了。

实际应用：

常用于多分类任务，但是 input 在输入 NLLLoss()之前，需要对 input 进行 log_softmax 函数激活，即将 input 转换成概率分布的形式，并且取对数。其实这些步骤在 CrossEntropyLoss 中就有，如果不想让网络的最后一层是 log_softmax 层的话，就可以采用 CrossEntropyLoss 完全代替此函数。

参数：

weight(Tensor)- 为每个类别的 loss 设置权值，常用于类别不均衡问题。weight 必须是 float 类型的 tensor，其长度要于类别 C 一致，即每一个类别都要设置有 weight。

size_average(bool)- 当 reduce=True 时有效。为 True 时，返回的 loss 为除以权重之和的平均值；为 False 时，返回的各样本的 loss 之和。

reduce(bool)- 返回值是否为标量，默认为 True。

ignore_index(int)- 忽略某一类别，不计算其 loss，其 loss 会为 0，并且，在采用 size_average 时，不会计算那一类的 loss，除的时候的分母也不会统计那一类的样本。

实例：

/Code/3_optimizer/3_1_lossFunction/4_NLLLoss.py

特别注意：

当带上权值，reduce = True, size_average = True，其计算公式为：

$$\ell(x, y) = \begin{cases} \sum_{n=1}^{N} \frac{1}{\sum_{n=1}^{N} w_{y_n}} l_n, & \text{if size\_average} = \text{True}, \\ \sum_{n=1}^{N} l_n, & \text{if size\_average} = \text{False}. \end{cases}$$

例如当 input 为[[0.6, 0.2, 0.2], [0.4, 1.2, 0.4]]，target= [0, 1], weight = [0.6, 0.2, 0.2]

l1 = - 0.6*0.6 = - 0.36

l2 = - 1.2*0.2 = - 0.24

loss = -0.36/(0.6+0.2) + -0.24/(0.6+0.2) = -0.75

5. PoissonNLLLoss

class **torch.nn.PoissonNLLLoss**(*log_input=True, full=False, size_average=None, eps=1e-08, reduce=None, reduction='elementwise_mean'*)

功能：

用于 target 服从泊松分布的分类任务。

计算公式：

$$\text{target} \sim \text{Poisson}(\text{input})$$
$$\text{loss}(\text{input}, \text{target}) = \text{input} - \text{target} * \log(\text{input}) + \log(\text{target!})$$

参数：

log_input(bool)- 为 True 时，计算公式为：loss(input,target)=exp(input) - target * input;

为 False 时，loss(input,target)=input - target * log(input+eps)

full(bool)- 是否计算全部的 loss。例如，当采用斯特林公式近似阶乘项时，此为

target*log(target) - target+0.5∗log(2πtarget)

eps(float)- 当 log_input = False 时，用来防止计算 log(0)，而增加的一个修正项。即

loss(input,target)=input - target * log(input+eps)

size_average(bool)- 当 reduce=True 时有效。为 True 时，返回的 loss 为平均值；为 False 时，返回的各样本的 loss 之和。

reduce(bool)- 返回值是否为标量，默认为 True

实例：

/Code/3_optimizer/3_1_lossFunction/5_PoissonNLLLoss.py

6. KLDivLoss

class **torch.nn.KLDivLoss**(*size_average=None, reduce=None, reduction='elementwise_mean'*)

功能：

计算 input 和 target 之间的 KL 散度(Kullback–Leibler divergence) 。

计算公式：

$$l(x, y) = L := \{l_1, \dots, l_N\}, \quad l_n = y_n \cdot (\log y_n - x_n)$$

（后面有代码手动计算，证明计算公式确实是这个，但是为什么没有对 x_n 计算对数呢？）

补充：KL 散度

KL 散度(Kullback–Leibler divergence) 又称为相对熵(Relative Entropy)，用于描述两个概率分布之间的差异。计算公式(离散时)：

$$D(p||q) = \sum_{i=1}^{n} p(x) \log \frac{p(x)}{q(x)}$$

其中 p 表示真实分布，q 表示 p 的拟合分布， D(P||Q)表示当用概率分布 q 来拟合真实分布 p 时，产生的信息损耗。这里的信息损耗，可以理解为损失，损失越低，拟合分布 q 越接近真实分布 p。同时也可以从另外一个角度上观察这个公式，即计算的是 p 与 q 之间的对数差在 p 上的期望值。

特别注意，D(p||q) ≠ D(q||p)， 其不具有对称性，因此不能称为 K-L 距离。

信息熵 = 交叉熵 - 相对熵

从信息论角度观察三者，其关系为信息熵 = 交叉熵 - 相对熵。在机器学习中，当训练数据固定，最小化相对熵 D(p||q) 等价于最小化交叉熵 H(p,q) 。

参数：

size_average(bool)- 当 reduce=True 时有效。为 True 时，返回的 loss 为平均值，平均值为 element-wise 的，而不是针对样本的平均；为 False 时，返回是各样本各维度的 loss 之和。

reduce(bool)- 返回值是否为标量，默认为 True。

使用注意事项：

要想获得真正的 KL 散度，需要如下操作：

1. reduce = True ； size_average=False

2. 计算得到的 loss 要对 batch 进行求平均

实例：

/Code/3_optimizer/3_1_lossFunction/6_KLDivLoss.py

7. BCELoss

class **torch.nn.BCELoss**(*weight=None, size_average=None, reduce=None, reduction='element wise_mean'*)

功能：

二分类任务时的交叉熵计算函数。此函数可以认为是 **nn.CrossEntropyLoss** 函数的特例。其分类限定为二分类，y 必须是{0,1}。还需要注意的是，input 应该为概率分布的形式，这样才符合交叉熵的应用。所以在 BCELoss 之前，input 一般为 sigmoid 激活层的输出，官方例子也是这样给的。该损失函数在自编码器中常用。

计算公式：

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_n \left[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n) \right]$$

参数：

weight(Tensor)- 为每个类别的 loss 设置权值，常用于类别不均衡问题。

size_average(bool)- 当 reduce=True 时有效。为 True 时，返回的 loss 为平均值；为 False 时，返回的各样本的 loss 之和。

reduce(bool)- 返回值是否为标量，默认为 True

8. BCEWithLogitsLoss

class **torch.nn.BCEWithLogitsLoss**(*weight=None, size_average=None, reduce=None, reduction='elementwise_mean', pos_weight=None*)

功能：

将 Sigmoid 与 BCELoss 结合，类似于 CrossEntropyLoss(将 nn.LogSoftmax()和 nn.NLLLoss()进行结合）。即 input 会经过 Sigmoid 激活函数，将 input 变成概率分布的形式。

计算公式：

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_n \left[t_n \cdot \log \sigma(x_n) + (1 - t_n) \cdot \log(1 - \sigma(x_n))\right]$$

σ() 表示 Sigmoid 函数

特别地，当设置 weight 时：

$$l_n = -w_n \left[p_n t_n \cdot \log \sigma(x_n) + (1 - t_n) \cdot \log(1 - \sigma(x_n))\right]$$

参数：

weight(Tensor)- ： 为 batch 中单个样本设置权值，If given, has to be a Tensor of size “nbatch”.

pos_weight-: 正样本的权重，当 p>1，提高召回率，当 P<1，提高精确度。可达到权衡召回率(Recall)和精确度(Precision)的作用。 Must be a vector with length equal to the number of classes.

size_average(bool)- 当 reduce=True 时有效。为 True 时，返回的 loss 为平均值；为 False 时，返回的各样本的 loss 之和。

reduce(bool)- 返回值是否为标量，默认为 True

9. MarginRankingLoss

class **torch.nn.MarginRankingLoss**(*margin=0, size_average=None, reduce=None, reduction='elementwise_mean'*)

功能：

计算两个向量之间的相似度，当两个向量之间的距离大于 margin，则 loss 为正，小于 margin，loss 为 0。

计算公式：

$$\text{loss}(x, y) = \max(0, -y * (x1 - x2) + \text{margin})$$

y == 1 时，x1 要比 x2 大，才不会有 loss，反之，y == -1 时，x1 要比 x2 小，才不会有 loss。

参数：

margin(float)- x1 和 x2 之间的差异。

size_average(bool)- 当 reduce=True 时有效。为 True 时，返回的 loss 为平均值；为 False 时，返回的各样本的 loss 之和。

reduce(bool)- 返回值是否为标量，默认为 True。

10. HingeEmbeddingLoss

class **torch.nn.HingeEmbeddingLoss**(*margin=1.0, size_average=None, reduce=None, reduction='elementwise_mean'*)

功能：

未知。为折页损失的拓展，主要用于衡量两个输入是否相似。 used for learning nonlinear embeddings or semi-supervised 。

计算公式：

$$l_n = \begin{cases} x_n, & \text{if } y_n = 1, \\ \max\{0, \Delta - x_n\}, & \text{if } y_n = -1, \end{cases}$$

参数：

margin(float)- 默认值为 1，容忍的差距。

size_average(bool)- 当 reduce=True 时有效。为 True 时，返回的 loss 为平均值；为 False 时，返回的各样本的 loss 之和。

reduce(bool)- 返回值是否为标量，默认为 True。

11. MultiLabelMarginLoss

class **torch.nn.MultiLabelMarginLoss**(*size_average=None, reduce=None, reduction='elementwise_mean'*)

功能：

用于一个样本属于多个类别时的分类任务。例如一个四分类任务，样本 x 属于第 0 类，第 1 类，不属于第 2 类，第 3 类。

计算公式：

$$\text{loss}(x, y) = \sum_{ij} \frac{\max(0, 1 - (x[y[j]] - x[i]))}{\text{x.size}(0)}$$

where $i == 0$ to $x.size(0)$, $j == 0$ to $y.size(0)$, $y[j] \geq 0$, and $i \neq y[j]$ for all i and j.

x[y[j]] 表示 样本 x 所属类的输出值，x[i]表示不等于该类的输出值。

参数：

size_average(bool)- 当 reduce=True 时有效。为 True 时，返回的 loss 为平均值；为 False 时，返回的各样本的 loss 之和。

reduce(bool)- 返回值是否为标量，默认为 True。

Input: (C) or (N,C) where N is the batch size and C is the number of classes.

Target: (C) or (N,C), same shape as the input.

12. SmoothL1Loss

class **torch.nn.SmoothL1Loss**(*size_average=None, reduce=None, reduction='elementwise_mean'*)

功能：

计算平滑 L1 损失，属于 Huber Loss 中的一种(因为参数 δ 固定为 1 了)。

补充：

Huber Loss 常用于回归问题，其最大的特点是对离群点（outliers）、噪声不敏感，具有较强的鲁棒性。

公式为：

$$L_\delta(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2 & \text{for} |y - f(x)| \le \delta, \\ \delta\,|y - f(x)| - \frac{1}{2}\delta^2 & \text{otherwise.} \end{cases}$$

理解为，当误差绝对值小于 δ，采用 L2 损失；若大于 δ，采用 L1 损失。

回到 SmoothL1Loss，这是 δ=1 时的 Huber Loss。

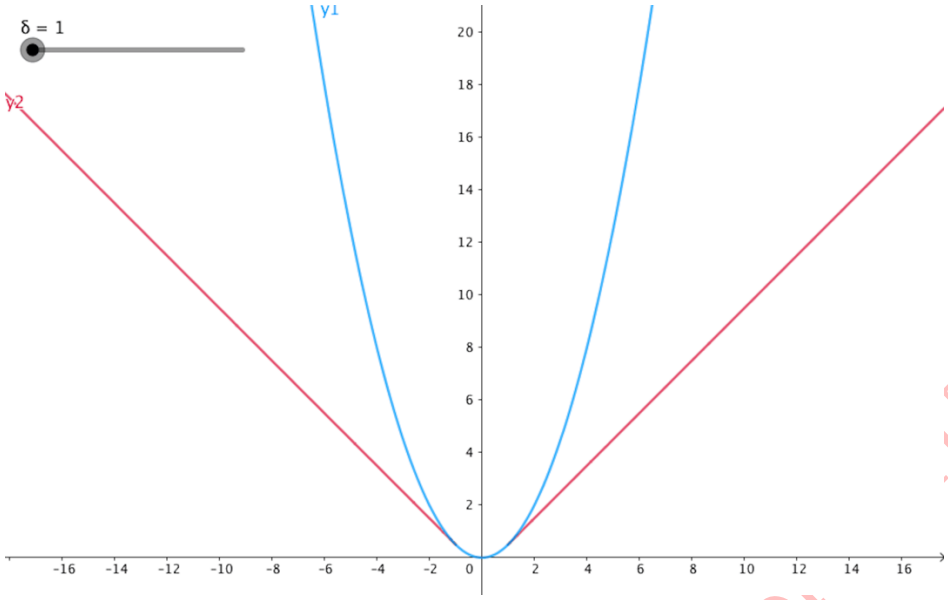
计算公式为：

$$\text{loss}(x, y) = \frac{1}{n}\sum_i z_i$$

where z_i is given by:

$$z_i = \begin{cases} 0.5(x_i - y_i)^2, & \text{if } |x_i - y_i| < 1 \\ |x_i - y_i| - 0.5, & \text{otherwise} \end{cases}$$

对应下图红色线：



参数：

size_average(bool)- 当 reduce=True 时有效。为 True 时，返回的 loss 为平均值；为 False 时，返回的各样本的 loss 之和。

reduce(bool)- 返回值是否为标量，默认为 True。

13. SoftMarginLoss

class **torch.nn.SoftMarginLoss**(*size_average=None, reduce=None, reduction='elementwise_mean'*)

功能：

Creates a criterion that optimizes a two-class classification logistic loss between input tensor xand target tensor y (containing 1 or -1). （暂时看不懂怎么用，有了解的朋友欢迎补充！）

计算公式：

$$\text{loss}(x, y) = \sum_i \frac{\log(1 + \exp(-y[i] * x[i]))}{\text{x.nelement()}}$$

参数：

size_average(bool)- 当 reduce=True 时有效。为 True 时，返回的 loss 为平均值；为 False 时，返回的各样本的 loss 之和。

reduce(bool)- 返回值是否为标量，默认为 True。

14. MultiLabelSoftMarginLoss

class **torch.nn.MultiLabelSoftMarginLoss**(*weight=None, size_average=None, reduce=None, reduction='elementwise_mean'*)

功能：

SoftMarginLoss 多标签版本，a multi-label one-versus-all loss based on max-entropy,

计算公式：

$$loss(x, y) = -\sum_i y[i] * \log((1 + \exp(-x[i]))^{-1}) + (1 - y[i]) * \log\left(\frac{\exp(-x[i])}{(1 + \exp(-x[i]))}\right)$$

参数：

weight(Tensor)- 为每个类别的 loss 设置权值。weight 必须是 float 类型的 tensor，其长度要于类别 C 一致，即每一个类别都要设置有 weight。

15. CosineEmbeddingLoss

class **torch.nn.CosineEmbeddingLoss**(*margin=0, size_average=None, reduce=None, reduction ='elementwise_mean'*)

功能：

用 Cosine 函数来衡量两个输入是否相似。 used for learning nonlinear embeddings or semi-supervised 。

计算公式：

$$\text{loss}(x, y) = \begin{cases} 1 - \cos(x_1, x_2), & \text{if } y == 1 \\ \max(0, \cos(x_1, x_2) - \text{margin}), & \text{if } y == -1 \end{cases}$$

参数：

margin(float)- ： 取值范围[-1,1]， 推荐设置范围 [0, 0.5]

size_average(bool)- 当 reduce=True 时有效。为 True 时，返回的 loss 为平均值；为 False 时，返回的各样本的 loss 之和。

reduce(bool)- 返回值是否为标量，默认为 True。

16. MultiMarginLoss

class **torch.nn.MultiMarginLoss**(*p=1, margin=1, weight=None, size_average=None, reduce=None, reduction='elementwise_mean'*)

功能：

计算多分类的折页损失。

计算公式：

$$\text{loss}(x, y) = \frac{\sum_i \max(0, w[y] * (\text{margin} - x[y] + x[i]))^p)}{\text{x.size}(0)}$$

其中，0≤y≤x.size(1) ； i == 0 to x.size(0) and i≠y; p==1 or p ==2; w[y]为各类别的weight。

参数：

p(int)- 默认值为 1，仅可选 1 或者 2。

margin(float)- 默认值为 1

weight(Tensor)- 为每个类别的 loss 设置权值。weight 必须是 float 类型的 tensor，其长度要于类别 C 一致，即每一个类别都要设置有 weight。

size_average(bool)- 当 reduce=True 时有效。为 True 时，返回的 loss 为平均值；为 False 时，返回的各样本的 loss 之和。

reduce(bool)- 返回值是否为标量，默认为 True。

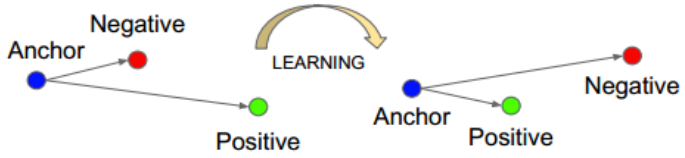
17. TripletMarginLoss

class **torch.nn.TripletMarginLoss**(*margin=1.0, p=2, eps=1e-06, swap=False, size_average=None, reduce=None, reduction='elementwise_mean'*)

功能：

计算三元组损失，人脸验证中常用。

如下图 Anchor、Negative、Positive，目标是让 Positive 元和 Anchor 元之间的距离尽可能的小，Positive 元和 Negative 元之间的距离尽可能的大。



从公式上看，Anchor 元和 Positive 元之间的距离加上一个 threshold 之后，要小于 Anchor 元与 Negative 元之间的距离。

$$\left\|x_i^a - x_i^p\right\|_2^2 + threshold < \left\|x_i^a - x_i^n\right\|_2^2$$

计算公式：

$$L(a,p,n) = \max\{d(a_i,p_i) - d(a_i,n_i) + \text{margin}, 0\}$$

$$\text{where } d(x_i,y_i) = \|\mathbf{x}_i - \mathbf{y}_i\|_p.$$

参数：

margin(float)- 默认值为 1

p(int)- The norm degree ， 默认值为 2

swap(float)– The distance swap is described in detail in the paper Learning shallow convolutional feature descriptors with triplet losses by V. Balntas, E. Riba et al. Default: False

size_average(bool)- 当 reduce=True 时有效。为 True 时，返回的 loss 为平均值；为 False 时，返回的各样本的 loss 之和。

reduce(bool)- 返回值是否为标量，默认为 True。

3.2 优化器基类：Optimizer

当数据、模型和损失函数确定，任务的数学模型就已经确定，接着就要选择一个合适的优化器(Optimizer)对该模型进行优化。

本节将介绍 PyTorch 中优化器的方法(函数)和使用方法，下一小节将介绍 PyTorch 中十种不同的优化器。

优化器基类 Optimizer

PyTorch 中所有的优化器(如：optim.Adadelta、optim.SGD、optim.RMSprop 等)均是 Optimizer 的子类，Optimizer 中定义了一些常用的方法，有 zero_grad()、step(closure)、state_dict()、load_state_dict(state_dict)和 add_param_group(param_group)，本节将会一一介绍。

3.2.1 参数组(param_groups)的概念

认识 Optimizer 的方法之前，需要了解一个概念，叫做参数组(param_groups)。在 finetune，某层定制学习率，某层学习率置零操作中，都会设计参数组的概念，因此首先了解参数组的概念非常有必要。

optimizer 对参数的管理是基于组的概念，可以为每一组参数配置特定的 lr,momentum,weight_decay 等等。

参数组在 optimizer 中表现为一个 list(self.param_groups)，其中每个元素是 dict，表示一个参数及其相应配置，在 dict 中包含'params'、'weight_decay'、'lr' 、'momentum'等字段。

实例：

/Code/3_optimizer/3_2_optimizer/1_param_groups.py

3.2.2 zero_grad()

功能：将梯度清零。

由于 PyTorch 不会自动清零梯度，所以在每一次更新前会进行此操作。

实例：

/Code/3_optimizer/3_2_optimizer/2_zero_grad.py

3.2.3 state_dict()

功能：获取模型当前的参数，以一个有序字典形式返回。

这个有序字典中，key 是各层参数名，value 就是参数。

实例：

/Code/3_optimizer/3_2_optimizer/3_state_dict.py

3.2.4 load_state_dict(state_dict)

功能：将 state_dict 中的参数加载到当前网络，常用于 finetune。

实例：

/Code/3_optimizer/3_2_optimizer/4_load_state_dict.py

3.2.5 add_param_group()

功能：给 optimizer 管理的参数组中增加一组参数，可为该组参数**定制 lr, momentum, weight_decay** 等，在 finetune 中常用。

例如：optimizer_1.add_param_group({'params': w3, 'lr': 0.001, 'momentum': 0.8})

实例：

/Code/3_optimizer/3_2_optimizer/5_add_param_group.py

3.2.6 step(closure)

功能：执行一步权值更新，其中可传入参数 closure（一个闭包）。如，当采用 LBFGS 优化方法时，需要多次计算，因此需要传入一个闭包去允许它们重新计算 loss

例如：

```
for input, target in dataset:
    def closure():
        optimizer.zero_grad()
        output = model(input)
        loss = loss_fn(output, target)
        loss.backward()
        return loss
```

```
optimizer.step(closure)
```

建议对照源码阅读，源码位于 torch/optim/optimizer.py: class Optimizer(object)

3.3 PyTorch 的十个优化器

上一节，介绍了所有优化器的基类——Optimizer 类，在 Optimizer 类中定义了 5 个实用的基本方法，虽然了解了 Optimizer 类，但是还无法构建一个能优化网络的优化器。

在本小节中，将会介绍目 PyTorch 提供的十种优化器，有常见的 SGD、ASGD、Rprop、RMSprop、Adam 等等。这里都说的是***优化器，并没有提***优化方法，因为 PyTorch 中给出的优化器与原始论文中提出的优化方法，多多少少有改动，详细还需看优化器源码。

1. torch.optim.SGD

class **torch.optim.SGD**(***params, lr=<object object>, momentum=0, dampening=0, weight_decay=0, nesterov=False***)

功能：

可实现 SGD 优化算法，带动量 SGD 优化算法，带 NAG(Nesterov accelerated gradient)动量 SGD 优化算法,并且均可拥有 weight_decay 项。

参数：

params(iterable)- 参数组(参数组的概念请查看 3.2 优化器基类：Optimizer)，优化器要管理的那部分参数。

lr(float)- 初始学习率，可按需随着训练过程不断调整学习率。

momentum(float)- 动量，通常设置为 0.9，0.8

dampening(float)- dampening for momentum ，暂时不了其功能，在源码中是这样用的：buf.mul_(momentum).add_(1 - dampening, d_p)，值得注意的是，若采用 nesterov，dampening 必须为 0.

weight_decay(float)- 权值衰减系数，也就是 L2 正则项的系数

nesterov(bool)- bool 选项，是否使用 NAG(Nesterov accelerated gradient)

注意事项：

pytroch 中使用 SGD 十分需要注意的是，更新公式与其他框架略有不同！

PyTorch 中是这样的：

$v=\rho*v+g$

$p=p-lr*v = p - lr*\rho*v - lr*g$

其他框架：

v=ρ*v+lr*g

p=p−v = p - ρ*v - lr*g

ρ 是动量，v 是速率，g 是梯度，p 是参数，其实差别就是在 ρ*v 这一项，PyTorch 中将此项也乘了一个学习率。

2. torch.optim.ASGD

***class* torch.optim.ASGD(*params, lr=0.01, lambd=0.0001, alpha=0.75, t0=1000000.0, weight_decay=0*)**

功能：

ASGD 也成为 SAG，均表示随机平均梯度下降(Averaged Stochastic Gradient Descent)，简单地说 ASGD 就是用空间换时间的一种 SGD，详细可参看论文：http://riejohnson.com/rie/stograd_nips.pdf

参数：

params(iterable)- 参数组(参数组的概念请查看 3.1 优化器基类：Optimizer)，优化器要优化的那些参数。

lr(float)- 初始学习率，可按需随着训练过程不断调整学习率。

lambd(float)- 衰减项，默认值 1e-4。

alpha(float)- power for eta update ，默认值 0.75。

t0(float)- point at which to start averaging，默认值 1e6。

weight_decay(float)- 权值衰减系数，也就是 L2 正则项的系数。

3. torch.optim.Rprop

***class* torch.optim.Rprop(*params, lr=0.01, etas=(0.5, 1.2), step_sizes=(1e-06, 50)*)**

功能：

实现 Rprop 优化方法(弹性反向传播)，优化方法原文《Martin Riedmiller und Heinrich Braun: Rprop - A Fast Adaptive Learning Algorithm. Proceedings of the International Symposium on Computer and Information Science VII, 1992》

该优化方法适用于 full-batch，不适用于 mini-batch，因而在 mini-batch 大行其道的时代里，很少见到。

4. torch.optim.Adagrad

class **torch.optim.Adagrad**(*params, lr=0.01, lr_decay=0, weight_decay=0, initial_accumulator_value=0*)

功能：

实现 Adagrad 优化方法(Adaptive Gradient)，Adagrad 是一种自适应优化方法，是自适应的为各个参数分配不同的学习率。这个学习率的变化，会受到梯度的大小和迭代次数的影响。梯度越大，学习率越小；梯度越小，学习率越大。缺点是训练后期，学习率过小，因为 Adagrad 累加之前所有的梯度平方作为分母。

详细公式请阅读：Adaptive Subgradient Methods for Online Learning and Stochastic Optimization

John Duchi, Elad Hazan, Yoram Singer; 12(Jul):2121—2159, 2011.(http://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf)

5. torch.optim.Adadelta

class **torch.optim.Adadelta**(*params, lr=1.0, rho=0.9, eps=1e-06, weight_decay=0*)

功能：

实现 Adadelta 优化方法。**Adadelta** 是 **Adagrad** 的改进。Adadelta 分母中采用距离当前时间点比较近的累计项，这可以避免在训练后期，学习率过小。

详细公式请阅读:https://arxiv.org/pdf/1212.5701.pdf

6. torch.optim.RMSprop

class **torch.optim.RMSprop**(*params, lr=0.01, alpha=0.99, eps=1e-08, weight_decay=0, momentum=0, centered=False*)

功能：

实现 RMSprop 优化方法（Hinton 提出），RMS 是均方根（root meam square）的意思。RMSprop 和 Adadelta 一样，也是对 Adagrad 的一种改进。RMSprop 采用均方根作为分母，可缓解 Adagrad 学习率下降较快的问题，并且引入均方根，可以减少摆动，详细了解可读：http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

7. torch.optim.Adam(AMSGrad)

class **torch.optim.Adam**(*params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False*)

功能：

实现 Adam(Adaptive Moment Estimation))优化方法。Adam 是一种自适应学习率的优化方法，Adam 利用梯度的一阶矩估计和二阶矩估计动态的调整学习率。吴老师课上说过，Adam 是结合了 Momentum 和 RMSprop，并进行了偏差修正。

参数：

amsgrad- 是否采用 AMSGrad 优化方法，asmgrad 优化方法是针对 Adam 的改进，通过添加额外的约束，使学习率始终为正值。(AMSGrad，ICLR-2018 Best-Pper 之一，《On the convergence of Adam and Beyond》)。

详细了解 Adam 可阅读,Adam: A Method for Stochastic Optimization(https://arxiv.org/abs/1412.6980)。

8. torch.optim.Adamax

class **torch.optim.Adamax**(*params, lr=0.002, betas=(0.9, 0.999), eps=1e-08, weight_decay=0*)

功能：

实现 Adamax 优化方法。Adamax 是对 Adam 增加了一个学习率上限的概念，所以也称之为 Adamax。

详细了解可阅读，Adam: A Method for Stochastic Optimization(https://arxiv.org/abs/1412.6980)(没错，就是 Adam 论文中提出了 Adamax)。

9. torch.optim.SparseAdam

class **torch.optim.SparseAdam**(*params, lr=0.001, betas=(0.9, 0.999), eps=1e-08*)

功能：

针对稀疏张量的一种“阉割版”Adam 优化方法。

- only moments that show up in the gradient get updated, and only those portions of the gradient get applied to the parameters

10. torch.optim.LBFGS

class **torch.optim.LBFGS**(*params, lr=1, max_iter=20, max_eval=None, tolerance_grad=1e-05, tolerance_change=1e-09, history_size=100, line_search_fn=None*)

功能：

实现 L-BFGS（Limited-memory Broyden－Fletcher－Goldfarb－Shanno）优化方法。L-BFGS 属于拟牛顿算法。L-BFGS 是对 BFGS 的改进，特点就是节省内存。

使用注意事项：

1.This optimizer doesn’t support per-parameter options and parameter groups (there can be only one).

2. Right now all parameters have to be on a single device. This will be improved in the future.(2018-10-07)

3.4 PyTorch 的六个学习率调整方法

上一小节对十种优化器进行了介绍，可以发现优化器中最重要的一个参数就是学习率，合理的学习率可以使优化器快速收敛。一般在训练初期给予较大的学习率，随着训练的进行，学习率逐渐减小。学习率什么时候减小，减小多少，这就涉及到学习率调整方法。

PyTorch 中提供了六种方法供大家使用，下面将一一介绍，最后对学习率调整方法进行总结。

1. lr_scheduler.StepLR

class **torch.optim.lr_scheduler.StepLR**(*optimizer, step_size, gamma=0.1, last_epoch=-1*)

功能：

等间隔调整学习率，调整倍数为 gamma 倍，调整间隔为 step_size。间隔单位是 step。需要注意的是，step 通常是指 epoch，不要弄成 iteration 了。

参数：

step_size(int)- 学习率下降间隔数，若为 30，则会在 30、60、90.....个 step 时，将学习率调整为 lr*gamma。

gamma(float)- 学习率调整倍数，默认为 0.1 倍，即下降 10 倍。

last_epoch(int)- 上一个 epoch 数，这个变量用来指示学习率是否需要调整。当 last_epoch 符合设定的间隔时，就会对学习率进行调整。当为-1 时，学习率设置为初始值。

2. lr_scheduler.MultiStepLR

class **torch.optim.lr_scheduler.MultiStepLR**(*optimizer, milestones, gamma=0.1, last_epoch=-1*)

功能：

按设定的间隔调整学习率。这个方法适合后期调试使用，观察 loss 曲线，为每个实验定制学习率调整时机。

参数：

milestones(list)- 一个 list，每一个元素代表何时调整学习率，list 元素必须是递增的。如 milestones=[30,80,120]

gamma(float)- 学习率调整倍数，默认为 0.1 倍，即下降 10 倍。

last_epoch(int)- 上一个 epoch 数，这个变量用来指示学习率是否需要调整。当 last_epoch 符合设定的间隔时，就会对学习率进行调整。当为-1 时，学习率设置为初始值。

3. lr_scheduler.ExponentialLR

class **torch.optim.lr_scheduler.ExponentialLR**(*optimizer, gamma, last_epoch=-1*)

功能：

按指数衰减调整学习率，调整公式：lr = lr * gamma**epoch

参数：

gamma- 学习率调整倍数的底，指数为 epoch，即 gamma**epoch

last_epoch(int)- 上一个 epoch 数，这个变量用来指示学习率是否需要调整。当 last_epoch 符合设定的间隔时，就会对学习率进行调整。当为-1 时，学习率设置为初始值。

4. lr_scheduler.CosineAnnealingLR

class **torch.optim.lr_scheduler.CosineAnnealingLR**(*optimizer, T_max, eta_min=0, last_epoch=-1*)

功能：

以余弦函数为周期，并在每个周期最大值时重新设置学习率。具体如下图所示

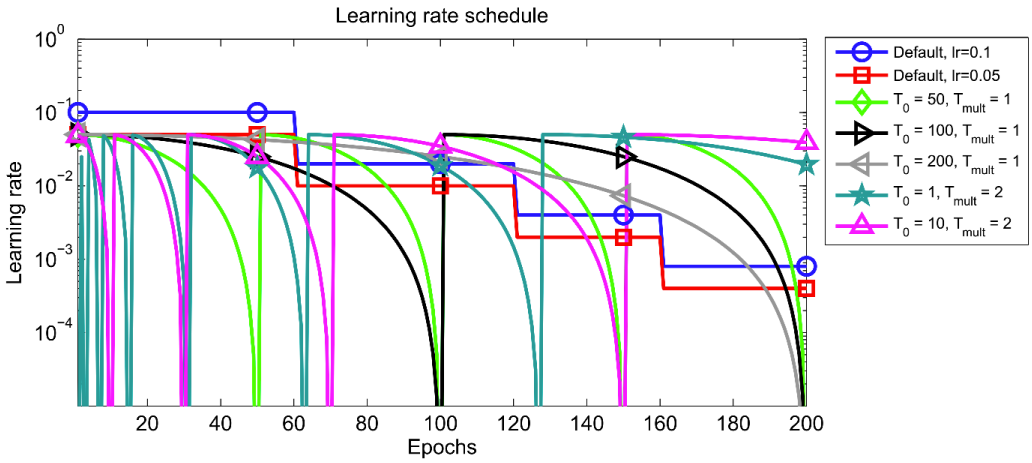


Figure 1: Alternative schedule schemes of learning rate η_t over batch index t: default schemes with $\eta_0 = 0.1$ (blue line) and $\eta_0 = 0.05$ (red line) as used by Zagoruyko & Komodakis (2016); warm restarts simulated every $T_0 = 50$ (green line), $T_0 = 100$ (black line) and $T_0 = 200$ (grey line) epochs with η_t decaying during i-th run from $\eta^i_{max} = 0.05$ to $\eta^i_{min} = 0$ according to eq. (5); warm restarts starting from epoch $T_0 = 1$ (dark green line) and $T_0 = 10$ (magenta line) with doubling ($T_{mult} = 2$) periods T_i at every new warm restart.

详细请阅读论文《 SGDR: Stochastic Gradient Descent with Warm Restarts》(ICLR-2017)：https://arxiv.org/abs/1608.03983

参数：

T_max(int)- 一次学习率周期的迭代次数，即 T_max 个 epoch 之后重新设置学习率。

eta_min(float)- 最小学习率，即在一个周期中，学习率最小会下降到 eta_min，默认值为 0。

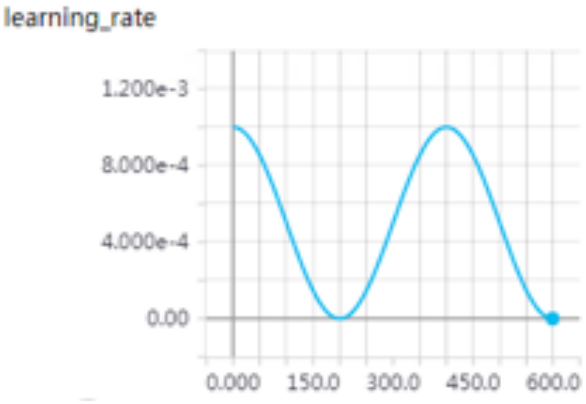
学习率调整公式为：

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})(1 + \cos(\frac{T_{cur}}{T_{max}}\pi))$$

可以看出是以初始学习率为最大学习率，以 2*Tmax 为周期，在一个周期内先下降，后上升。

实例：

T_max = 200, 初始学习率 = 0.001, eta_min = 0



5. lr_scheduler.ReduceLROnPlateau

class **torch.optim.lr_scheduler.ReduceLROnPlateau**(***optimizer, mode='min', factor=0.1, patience=10, verbose=False, threshold=0.0001, threshold_mode='rel', cooldown=0, min_lr=0, eps=1e-08***)

功能：

当某指标不再变化（下降或升高），调整学习率，这是非常实用的学习率调整策略。例如，当验证集的 loss 不再下降时，进行学习率调整；或者监测验证集的 accuracy，当 accuracy 不再上升时，则调整学习率。

参数：

mode(str)- 模式选择，有 min 和 max 两种模式，min 表示当指标不再降低(如监测 loss)，max 表示当指标不再升高(如监测 accuracy)。

factor(float)- 学习率调整倍数(等同于其它方法的 gamma)，即学习率更新为 lr = lr * factor

patience(int)- 直译——"耐心"，即忍受该指标多少个 step 不变化，当忍无可忍时，调整学习率。

verbose(bool)- 是否打印学习率信息， print('Epoch {:5d}: reducing learning rate' ' of group {} to {:.4e}.'.format(epoch, i, new_lr))

threshold(float)- Threshold for measuring the new optimum，配合 threshold_mode 使用。

threshold_mode(str)- 选择判断指标是否达最优的模式，有两种模式，rel 和 abs。

当 threshold_mode==rel，并且 mode==max 时，dynamic_threshold = best * (1 + threshold)；

当 threshold_mode==rel，并且 mode==min 时，dynamic_threshold = best * (1 - threshold)；

当 threshold_mode==abs，并且 mode==max 时，dynamic_threshold = best + threshold ；

当 threshold_mode==rel，并且 mode==max 时，dynamic_threshold = best - threshold

cooldown(int)- “冷却时间“，当调整学习率之后，让学习率调整策略冷静一下，让模型再训练一段时间，再重启监测模式。

min_lr(float or list)- 学习率下限，可为 float，或者 list，当有多个参数组时，可用 list 进行设置。

eps(float)- 学习率衰减的最小值，当学习率变化小于 eps 时，则不调整学习率。

6. lr_scheduler.LambdaLR

class **torch.optim.lr_scheduler.LambdaLR(*optimizer, lr_lambda, last_epoch=-1*)**

功能：

为不同参数组设定不同学习率调整策略。调整规则为，lr = base_lr * lmbda(self.last_epoch) 。

参数：

lr_lambda(function or list)- 一个计算学习率调整倍数的函数，输入通常为 step，当有多个参数组时，设为 list。

last_epoch(int)- 上一个 epoch 数，这个变量用来指示学习率是否需要调整。当 last_epoch 符合设定的间隔时，就会对学习率进行调整。当为-1 时，学习率设置为初始值。

例如：

```
ignored_params = list(map(id, net.fc3.parameters()))
base_params = filter(lambda p: id(p) not in ignored_params, net.parameters())
optimizer = optim.SGD([
{'params': base_params},
{'params': net.fc3.parameters(), 'lr': 0.001*100}], 0.001, momentum=0.9,
weight_decay=1e-4)
lambda1 = lambda epoch: epoch // 3
lambda2 = lambda epoch: 0.95 ** epoch
```

```
scheduler = torch.optim.lr_scheduler.LambdaLR(optimizer, lr_lambda=[lambda1, lambda2])
for epoch in range(100):
scheduler.step()
print('epoch: ', i, 'lr: ', scheduler.get_lr())
train(...)
validate(...)
```

输出：

```
epoch: 0 lr: [0.0, 0.1]
epoch: 1 lr: [0.0, 0.095]
epoch: 2 lr: [0.0, 0.09025]
epoch: 3 lr: [0.001, 0.0857375]
epoch: 4 lr: [0.001, 0.081450625]
epoch: 5 lr: [0.001, 0.07737809374999999]
epoch: 6 lr: [0.002, 0.07350918906249998]
epoch: 7 lr: [0.002, 0.06983372960937498]
epoch: 8 lr: [0.002, 0.06634204312890622]
epoch: 9 lr: [0.003, 0.0630249409724609]
```

为什么第一个参数组的学习率会是 0 呢？ 来看看学习率是如何计算的。

第一个参数组的初始学习率设置为 0.001，lambda1 = lambda epoch: epoch // 3，

第 1 个 epoch 时，由 lr = base_lr * lmbda(self.last_epoch)，可知道 lr = 0.001 * (0//3) ，又因为 1//3 等于 0，所以导致学习率为 0。

第二个参数组的学习率变化，就很容易看啦，初始为 0.1，lr = 0.1 * 0.95^epoch ，当 epoch 为 0 时，lr=0.1 ，epoch 为 1 时，lr=0.1*0.95。

学习率调整小结

PyTorch 提供了六种学习率调整方法，可分为三大类，分别是

1. 有序调整；
2. 自适应调整；
3. 自定义调整。

第一类，依一定规律有序进行调整，这一类是最常用的，分别是等间隔下降(Step)，按需设定下降间隔(MultiStep)，指数下降(Exponential)和 CosineAnnealing。这四种方法的调整时机都是人为可控的，也是训练时常用到的。

第二类，依训练状况伺机调整，这就是 ReduceLROnPlateau 方法。该法通过监测某一指标的变化情况，当该指标不再怎么变化的时候，就是调整学习率的时机，因而属于自适应的调整。

第三类，自定义调整，Lambda。Lambda 方法提供的调整策略十分灵活，我们可以为不同的层设定不同的学习率调整方法，这在 fine-tune 中十分有用，我们不仅可为不同的层设定不同的学习率，还可以为其设定不同的学习率调整策略，简直不能更棒！

step 源码阅读

在 PyTorch 中，学习率的更新是通过 scheduler.step()，而我们知道影响学习率的一个重要参数就是 epoch，而 epoch 与 scheduler.step()是如何关联的呢？这就需要看源码了。

源码在 torch/optim/lr_scheduler.py，step()方法在_LRScheduler 类当中，该类作为所有学习率调整的基类，其中定义了一些基本方法，如现在要介绍的 step()，以及最常用的 get_lr()，不过 get_lr()是一个虚函数，均需要在派生类中重新定义函数。

看看 step()

```
def step(self, epoch=None):
if epoch is None:
epoch = self.last_epoch + 1
self.last_epoch = epoch
for param_group, lr in zip(self.optimizer.param_groups, self.get_lr()):
param_group['lr'] = lr
```

函数接收变量 epoch，默认为 None，当为 None 时，epoch = self.last_epoch + 1。从这里知道，last_epoch 是用以记录 epoch 的。上面有提到 last_epoch 的初始值是-1，因此，第一个 epoch 的值为 -1+1 =0。接着最重要的一步就是获取学习率，并更新。

由于 PyTorch 是基于参数组的管理方式，这里需要采用 for 循环对每一个参数组的学习率进行获取及更新。这里需要注意的是 get_lr()，get_lr()的功能就是获取当前 epoch，该参数组的学习率。

这里以 StepLR()为例，介绍 get_lr()，请看代码：

```
def get_lr(self):
return [base_lr * self.gamma ** (self.last_epoch // self.step_size) for base_lr
in self.base_lrs]
```

由于 PyTorch 是基于参数组的管理方式，可能会有多个参数组，因此用 for 循环，返回的是一个 list。list 元素的计算方式为

base_lr * self.gamma ** (self.last_epoch // self.step_size)。

看完代码，可以知道，在执行一次 scheduler.step()之后，epoch 会加 1，因此 scheduler.step()要放在 epoch 的 for 循环当中执行。

第四章 监控模型——可视化

神经网络是一个复杂的数学模型，很多东西暂时没办法解释。但归根到底，它始终是一个数学模型，我们就可以用统计的方法去观察它，理解它。

本章将介绍如何在 PyTorch 中使用 TensorBoardX 对神经网络进行统计可视化，如 Loss 曲线、Accuracy 曲线、卷积核可视化、权值直方图及多分位数折线图、特征图可视化、梯度直方图及多分位数折线图及混淆矩阵图等。

4.1 TensorBoardX

PyTorch 自身的可视化功能没有 TensorFlow 的 tensorboard 那么优秀，所以 PyTorch 通常是借助 tensorboard(是借助，非直接使用)进行可视化，目前流行的有如下两种方法，本文仅介绍第二种——TensorBoardX。

1. 构建 Logger 类

Logger 类中“包”了 tf.summary.FileWriter，截至目前（2018.10.17），只有三种操作，分别是 scalar_summary()，image_summary()，histo_summary()。

优点：轻便，可满足大部分需求

Logger 类参考 github：https://github.com/yunjey/PyTorch-tutorial/tree/master/tutorials/04-utils/tensorboard

2. 借助 TensorBoardX 包

TensorBoardX 包的功能就比较全，截至目前（2018.10.17），支持除 tensorboard beholder 之外的所有 tensorboard 的记录类型。

github：https://github.com/lanpa/tensorboardX

API 文档：https://tensorboard-PyTorch.readthedocs.io/en/latest/tutorial_zh.html#

安装时小插曲：

一开始按照 github 上的方法安装：pip install git+https://github.com/lanpa/tensorboardX

会报错：from .proto importevent_pb2 ImportError: cannot import name event_pb2

查看本地 在文件夹 proto/下确实没有 event_pb2,但是在 github 上是存在的。

最后通过 pip uninstall tensorboardX，使用 pip install tensorboard 安装成功。

tensorboardX 最早叫 tensorboard，但此名易引起混淆，之后改为 tensorboardX，which stands for tensorboard for X。

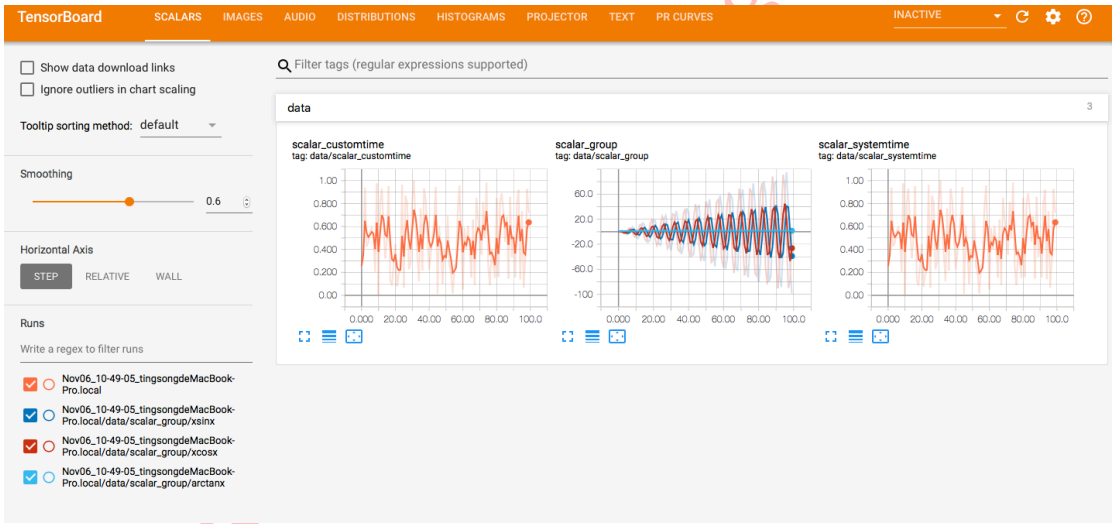
代码实现：

tensorboardX 提供 13 个函数，可以记录标量、图像、语音、文字等等，功能十分丰富。本节将对这些函数进行介绍，所用代码为 tensorboardX 的官方 demo.py，放在 **/Code/4_viewer/1_tensorboardX_demo.py**，请先运行该文件，并且打开 terminal，进入相应的虚拟环境（如果有），进入到/Result/文件夹，

执行：tensorboard --logdir=runs

然后到浏览器中打开：localhost:6006

可以看到显示界面如下：



完成以上步骤，就可以一一来学习 tensorbaordX 各功能函数啦。

tesnorboardX 的函数：

1. add_scalar()

add_scalar(*tag, scalar_value, global_step=None, walltime=None*)

功能：

在一个图表中记录一个标量的变化，常用于 Loss 和 Accuracy 曲线的记录。

参数：

tag(string)- 该图的标签，类似于 polt.title。

scalar_value(float or string/blobname)- 用于存储的值，曲线图的 y 坐标

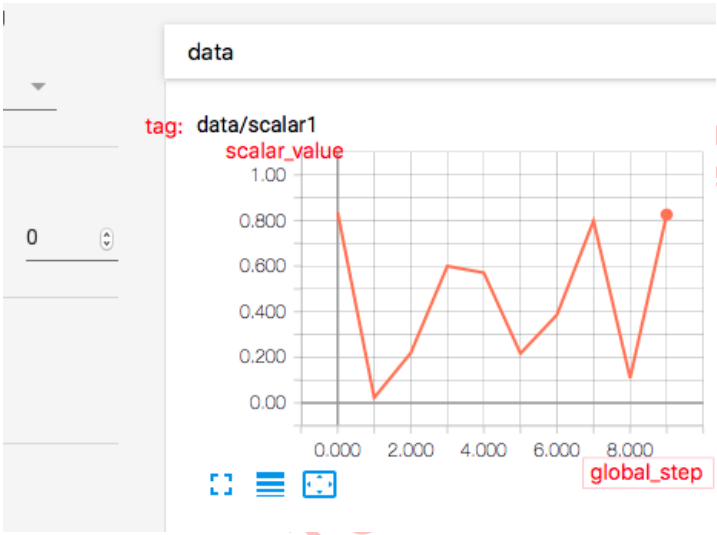
global_step(int)- 曲线图的 x 坐标

walltime(float)- 为 event 文件的文件名设置时间，默认为 time.time()

运行 demo 中的：

用 github 首页 demo 运行这一行：writer.add_scalar('data/scalar1', dummy_s1[0], n_iter)

可以得到下图：



2. add_scalars()

add_scalars(*main_tag, tag_scalar_dict, global_step=None, walltime=None*)

功能：

在一个图表中记录多个标量的变化，常用于对比，如 trainLoss 和 validLoss 的比较等。

参数：

main_tag(string)- 该图的标签。

tag_scalar_dict(dict)- key 是变量的 tag，value 是变量的值。

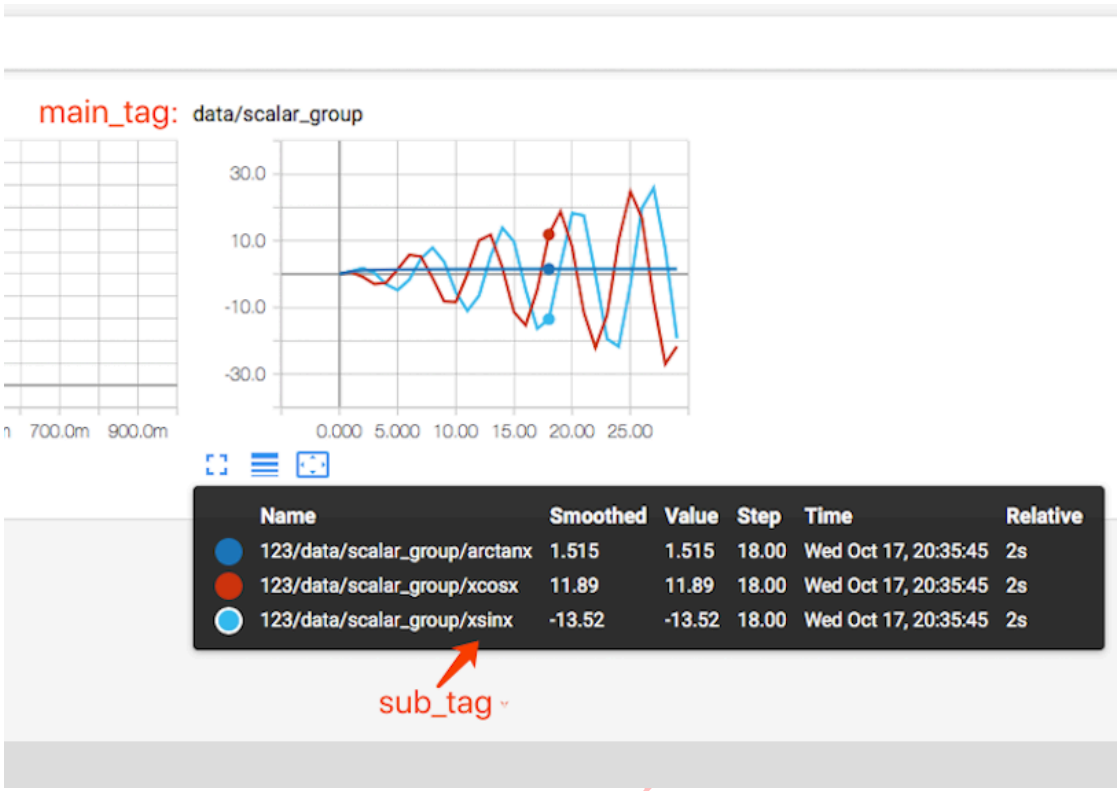
global_step(int)- 曲线图的 x 坐标

walltime(float)- 为 event 文件的文件名设置时间，默认为 time.time()

运行 demo 中的：

```
writer.add_scalars('data/scalar_group', {'xsinx': n_iter * np.sin(n_iter),
'xcosx': n_iter * np.cos(n_iter),
'arctanx': np.arctan(n_iter)}, n_iter)
```

可以得到下图：



3. add_histogram()

add_histogram(*tag, values, global_step=None, bins='tensorflow', walltime=None*)

功能：

绘制直方图和多分位数折线图，常用于监测权值及梯度的分布变化情况，便于诊断网络更新方向是否正确。

参数：

tag(string)- 该图的标签，类似于 polt.title。

values(torch.Tensor, numpy.array or string/blobname)- 用于绘制直方图的值

global_step(int)- 曲线图的 y 坐标

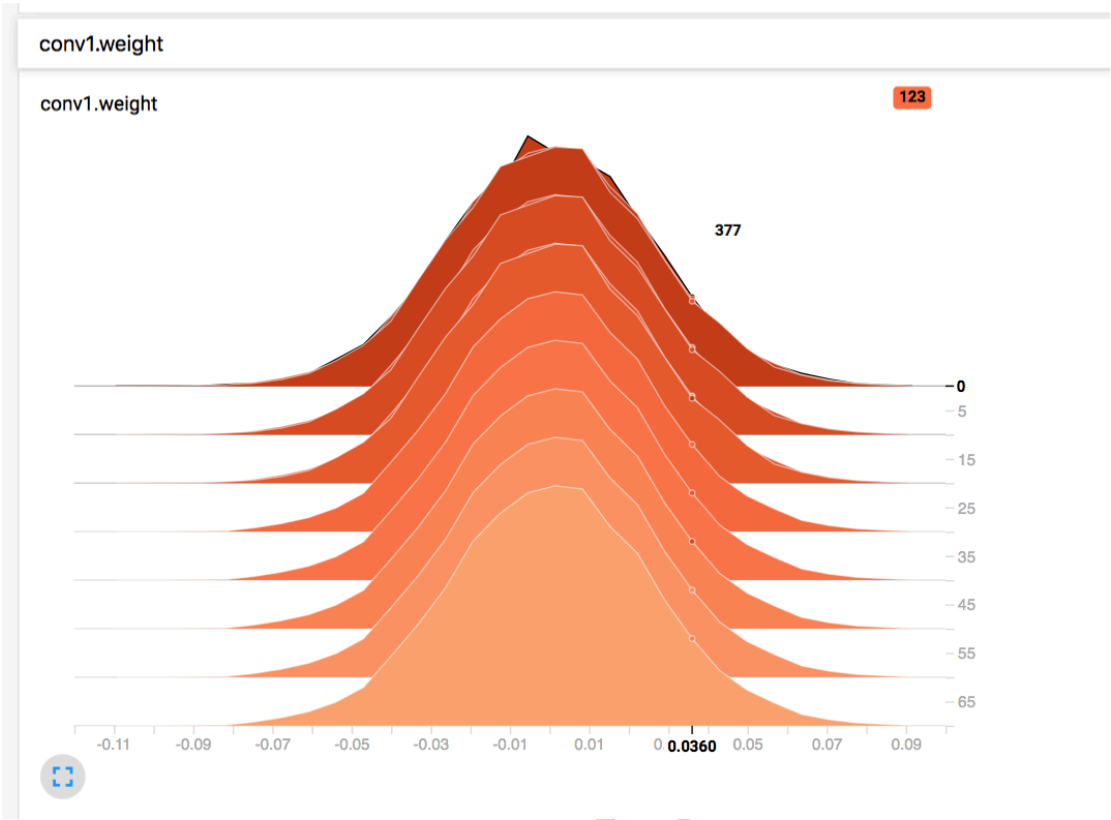
bins(string)- 决定如何取 bins，默认为‘tensorflow’，可选：’auto’，‘fd’等

walltime(float)- 为 event 文件的文件名设置时间，默认为 time.time()

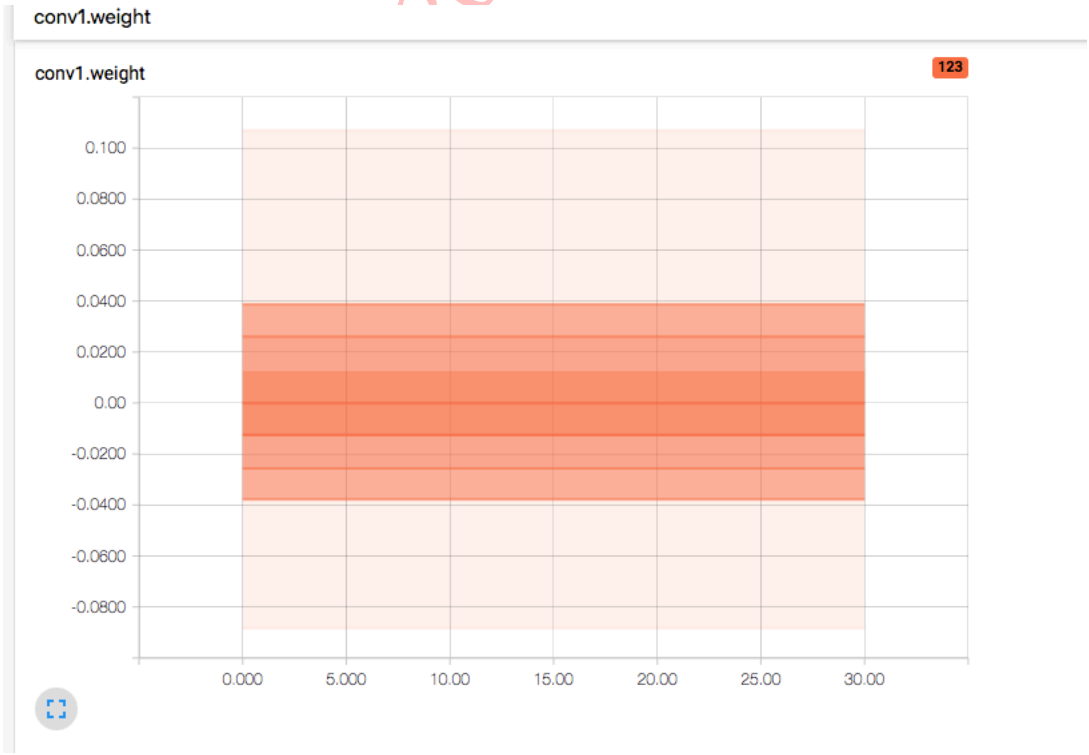
运行 demo 中的：

```
for name, param in resnet18.named_parameters():
writer.add_histogram(name, param.clone().cpu().data.numpy(), n_iter)
```

可以得到以下两种图分别在 HISTOGRAMS 和 DISTRIBUTIONS 里面：



x 轴即变量大小，y 轴为 gloabl_step。377 表示卷积层 conv1 的权值中有 377 个 weight 的大小是在 0.036 这个区间。



x 轴为 gloabl_step，由于这里没有训练，所以随着 x 轴的增加，曲线是平直的。看 y 轴，从上到下，共计 9 条曲线(若有训练，会是曲线，现在是直线)，分别对应[maximum, 93%, 84%, 69%, 50%, 31%, 16%, 7%, minimum]分位数。

4. add_image()

add_image(*tag, img_tensor, global_step=None, walltime=None*)

功能：

绘制图片，可用于检查模型的输入，监测 feature map 的变化，或是观察 weight。

参数：

tag(string)- 该图的标签，类似于 polt.title。

img_tensor(torch.Tensor,numpy.array, or string/blobname)- 需要可视化的图片数据， shape = [C,H,W]。

global_step(int)- x 坐标。

walltime(float)- 为 event 文件的文件名设置时间，默认为 time.time()。

通常会借助 torchvision.utils.make_grid() 将一组图片绘制到一个窗口

补充 torchvision.utils.make_grid()

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

功能：

将一组图片拼接成一张图片，便于可视化。

参数：

tensor(Tensor or list)- 需可视化的数据，shape:(B x C x H x W),B 表示 batch 数，即几张图片

nrow(int)- 一行显示几张图，默认值为 8。

padding(int)- 每张图片之间的间隔，默认值为 2。

normalize(bool)- 是否进行归一化至(0,1)。

range(tuple)- 设置归一化的 min 和 max，若不设置，默认从 tensor 中找 min 和 max。

scale_each(bool)- 每张图片是否单独进行归一化，还是 min 和 max 的一个选择。

pad_value(float)- 填充部分的像素值，默认为 0，即黑色。

运行 demo 代码：

```
import torchvision.utils as vutils
dummy_img = torch.rand(32, 3, 64, 64) # (B x C x H x W)
if n_iter % 10 == 0:
x = vutils.make_grid(dummy_img, normalize=True, scale_each=True)
writer.add_image('Image', x, n_iter) # x.size= (3, 266, 530) (C*H*W)
```

可得到下图：

5. add_graph()

add_graph(*model, input_to_model=None, verbose=False, **kwargs*)

功能：

绘制网络结构拓扑图。

参数：

model(torch.nn.Module)- 模型实例

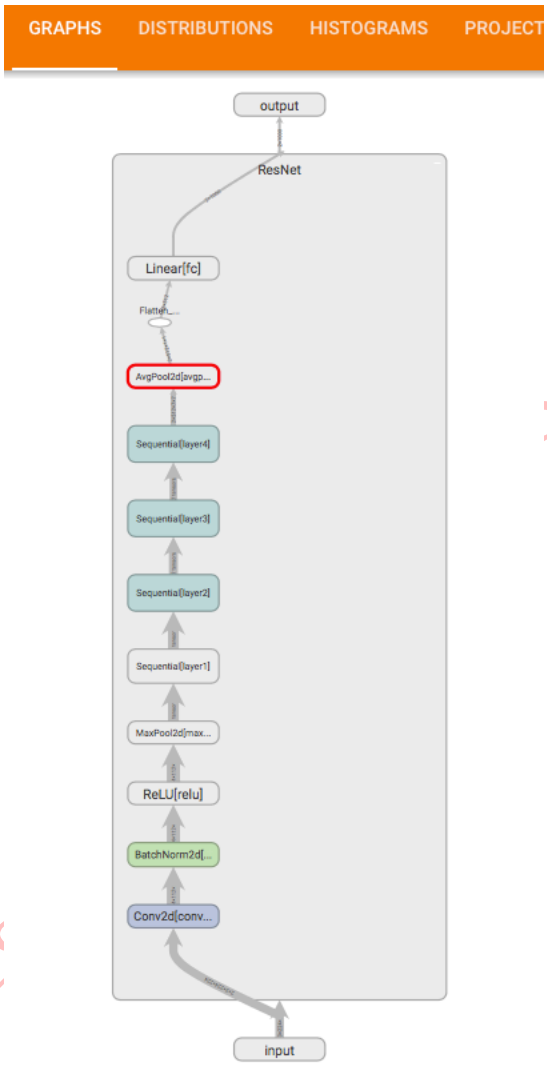
inpjt_to_model(torch.autograd.Variable)- 模型的输入数据，可以生成一个随机数，只要 shape 符合要求即可

运行以下代码：

```
import torchvision.models as models
```

```
resnet18 = models.resnet18(False)
dummy_input = torch.rand(6, 3, 224, 224)
writer.add_graph(resnet18, dummy_input)
```

可在 GRAPHS 中看到 Resnet18 的网络拓扑



6. add_embedding()

add_embedding(*mat, metadata=None, label_img=None, global_step=None, tag='default', metadata_header=None*)

功能：

在三维空间或二维空间展示数据分布，可选 T-SNE、PCA 和 CUSTOM 方法。

参数：

mat(torch.Tensor or numpy.array)- 需要绘制的数据，一个样本必须是一个向量形式。

shape = (N,D)，N 是样本数，D 是特征维数。

metadata(list)- 数据的标签，是一个 list，长度为 N。

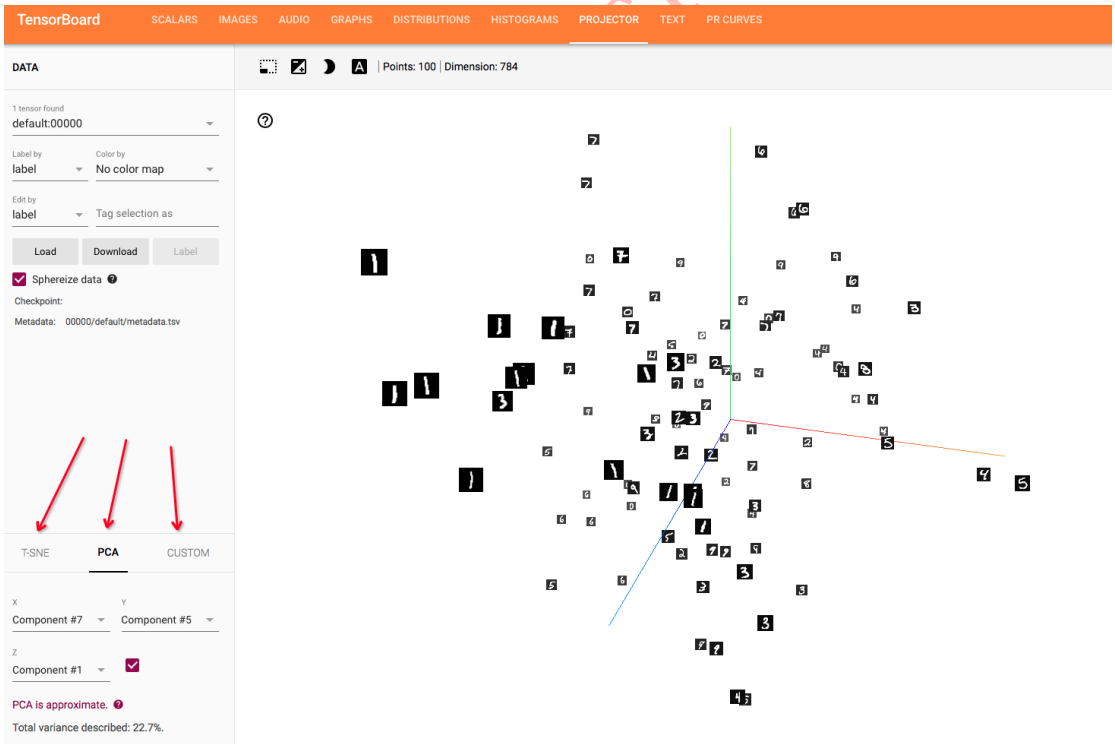
label_img(torch.Tensor)- 空间中展示的图片，shape = (N,C,H,W)。

global_step(int)- Global step value to record，不理解这里有何用处呢？知道的朋友补充一下吧。

tag(string)- 标签

以下代码，展示 Mnist 中的 100 张图片

```
dataset = datasets.MNIST('mnist', train=False, download=True)
images = dataset.test_data[:100].float()
label = dataset.test_labels[:100]
features = images.view(100, 784)
writer.add_embedding(features, metadata=label, label_img=images.unsqueeze(1))
```



7. add_text()

add_text(*tag, text_string, global_step=None, walltime=None*)

功能：记录文字

8. add_video()

add_video(*tag, vid_tensor, global_step=None, fps=4, walltime=None*)

功能：记录 video

9. add_figure()

add_figure(*tag, figure, global_step=None, close=True, walltime=None*)

功能：添加 matplotlib 图片到图像中

10. add_image_with_boxes()

add_image_with_boxes(*tag, img_tensor, box_tensor, global_step=None, walltime=None, **kwargs*)

功能：图像中绘制 Box，目标检测中会用到

11. add_pr_curve()

add_pr_curve(*tag, labels, predictions, global_step=None, num_thresholds=127, weights=None, walltime=None*)

功能：绘制 PR 曲线

12. add_pr_curve_raw()

add_pr_curve_raw(*tag, true_positive_counts, false_positive_counts, true_negative_counts, false_negative_counts, precision, recall, global_step=None, num_thresholds=127, weights=None, walltime=None*)

功能：从原始数据上绘制 PR 曲线

13. export_scalars_to_json()

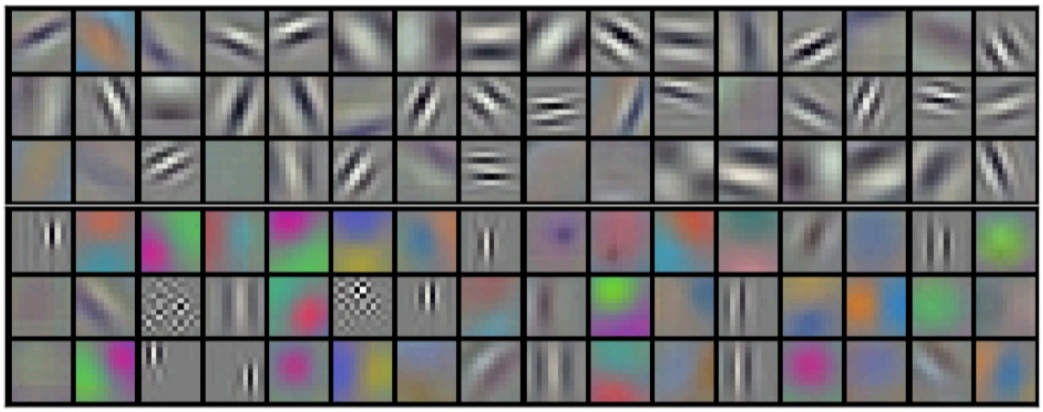
export_scalars_to_json(*path*)

功能：将 scalars 信息保存到 json 文件，便于后期使用

4.2 卷积核可视化

神经网络中最重要的就是权值，而人们对神经网络理解有限，所以我们需要通过尽可能了解权值来帮助诊断网络的训练情况。除了查看权值分布图和多折线分位图，还可以对卷积核权值进行可视化，来辅助我们分析网络。本小节就介绍卷积核权值可视化原理和方法。

在 2012 年 AlexNet 的论文中，展示了一副卷积核权值可视化的图片，如下图所示。



文中将第一个卷积层的卷积核权值进行可视化，发现有趣的现象。第一个 GPU 中的卷积核呈现初边缘的特性，第二个 GPU 中的卷积核呈现色彩的特性。对卷积核权值进行可视化，在一定程度上帮助我们诊断网络的训练好坏，因此对卷积核权值的可视化十分有必要。

可视化原理很简单，对单个卷积核进行“归一化”至 0～255，然后将其展现出来即可，这一系列操作可以借助 TensorboardX 的 add_image 来实现。

先看两张可视化效果图，接着简单介绍卷积核卷积过程，最后讲解代码。

下图以卷积层 conv1 为例，conv1.weight.shape() = [6,3,5,5]，输入通道数为 3，卷积核个数为 6，则 feature map 数为 6，卷积核大小为 5*5。

图 1，绘制全部卷积核。

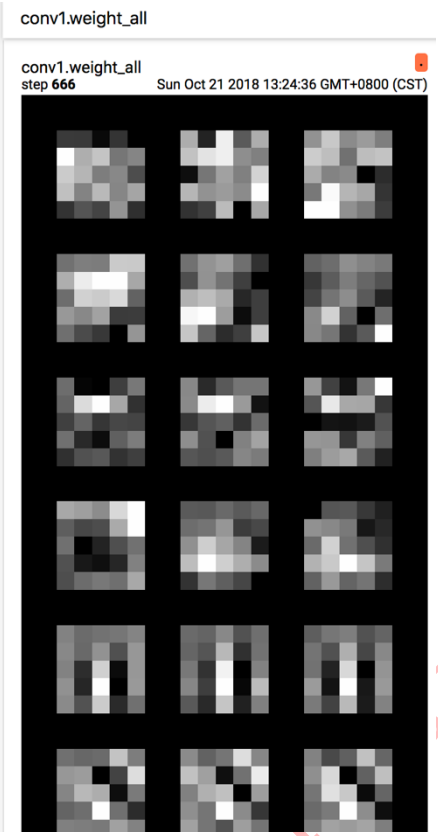
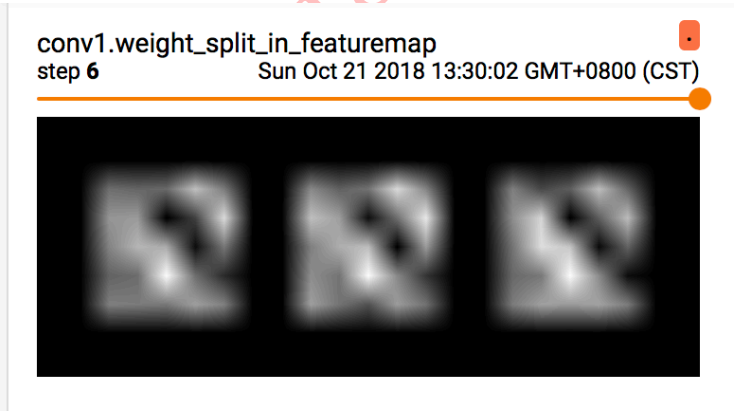


图 2，以 feature map 为单位，借助 step 参数区分 feature map，将卷积核分开绘制。

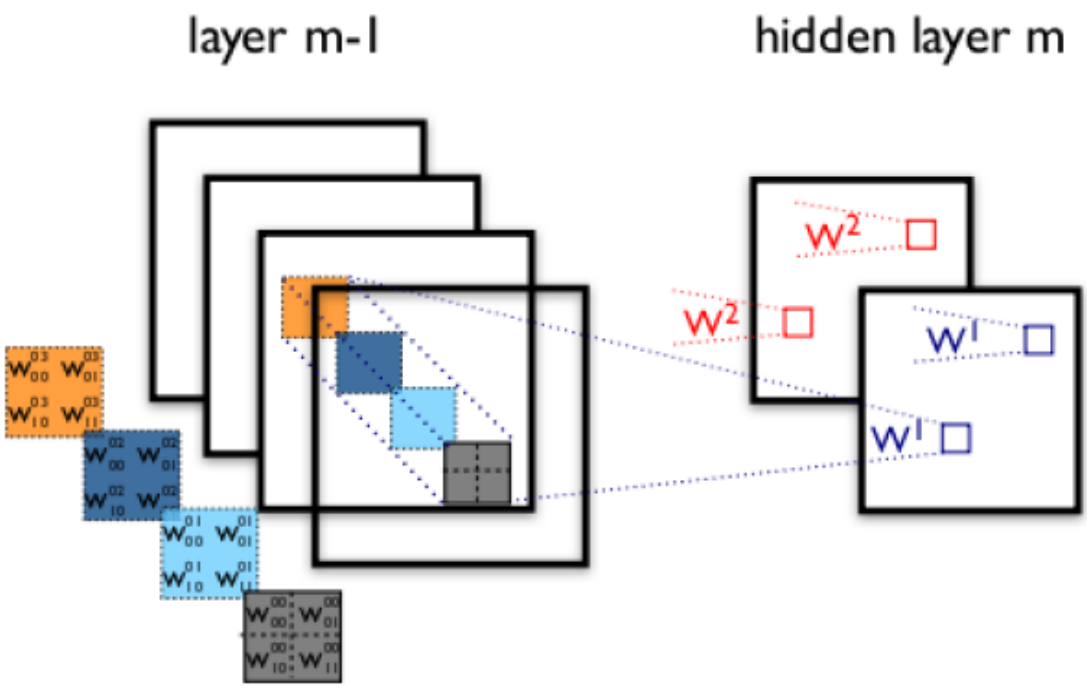


好奇的朋友就会问了，为什么要将每一行显示 3 个卷积核呢？

因为每 3 个卷积核共同作用去决定了一张 feature map，因此将其放在一起观察。具体卷积核过程可以观察下图，输入通道数为 4，输出通道数为 2，卷积层卷积核有 8 个 2*2 的卷积核。

第一个输出特征图的值，是由 4 个卷积核分别对 4 个输入通道进行卷积并求和，加上 bias 项，再通过激活函数，才得到最终 feature map 上的值。

所以可以将卷积层的卷积核按照 feature map 输出数进行划分，一个 feature map 对应的卷积核数为输入通道数。



代码实现：

运行 /Code/4_viewer/2_visual_weights.py

运行代码后，在文件夹/Result/下获得文件夹 visual_weights，然后打开 terminal，进入相应的虚拟环境（如果有），进入目录/Result/

执行：tensorboard --logdir=visual_weights

进入浏览器，打开网页：http://localhost:6006/

4.3 特征图可视化

有时候我们会好奇，原始图像经过网络的操作之后，会是什么样。本小节就介绍如何可视化经网络操作后的图像(feature maps)。

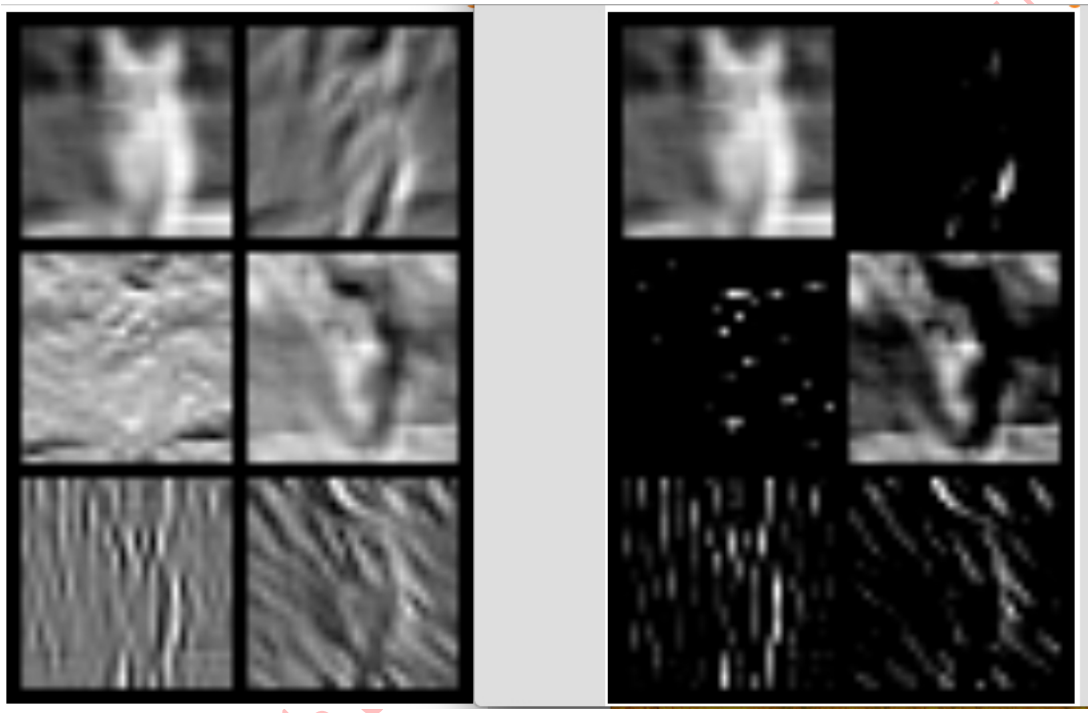
基本思路：

1. 获取图片，将其转换成模型输入前的数据格式，即一系列 transform，
2. 获取模型各层操作，手动的执行每一层操作，拿到所需的 feature maps，
3. 借助 tensorboardX 进行绘制。

Tips：

1. 此处获取模型各层操作是__init__()中定义的操作，然而模型真实运行采用的是forward()，所以需要人工比对两者差异。本例的差异是，__init__()中缺少激活函数relu。

先看看图，下图为 conv1 层输出的 feature maps， 左图为未经过 relu 激活函数，右图为经过 relu 之后的 feature maps。



原始图片(已经 resize 至 32*32)

代码实现：

运行 /Code/4_viewer/3_visual_featuremaps.py

运行代码后，在文件夹/Result/下获得文件夹 visual_featuremaps，然后打开 terminal，进入相应的虚拟环境（如果有），进入目录/Result/

执行：tensorboard --logdir=visual_featuremaps

进入浏览器，打开网页：http://localhost:6006/

文末探讨：

1. 经过 relu 激活函数之后，明显很多像素点变黑，即像素值变为 0 了？ 这样是否会影响网络性能？

2. 除了 net._modules.items()这种方法之外，是否还有更简便的方法来获取 feature maps 吗？ 因此上述方法感觉不太“智能”，需要人工比对 forward()中的操作和 __init__()中的操作，比较容易出差错，如果有更好的方法，欢迎分享。

4.4 梯度及权值分布可视化

在网络训练过程中，我们常常会遇到梯度消失、梯度爆炸等问题，我们可以通过记录每个 epoch 的梯度的值来监测梯度的情况，还可以记录权值，分析权值更新的方向是否符合规律。

本小节就介绍如何记录梯度及权值，并进行可视化。

代码实现：

运行 /Code/4_viewer/4_hist_grad_weight.py

运行代码后，在文件夹/Result/下获得文件夹 hist_grad_weight，然后打开 terminal，进入相应的虚拟环境（如果有），进入目录/Result/

执行：tensorboard --logdir=hist_grad_weight

进入浏览器，打开网页：http://localhost:6006/

记录梯度和权值主要是以下三行代码：

```
# 每个 epoch，记录梯度，权值
for name, layer in net.named_parameters():
writer.add_histogram(name + '_grad', layer.grad.cpu().data.numpy(), epoch)
writer.add_histogram(name + '_data', layer.cpu().data.numpy(), epoch)
```

可视化分析：

1. 权值 weights 的监控

经过 100 个 epoch 的训练，来看看第一个卷积层的权值分布的变化。

x 轴即变量大小，y 轴为 gloabl_step。图 1 x=0.306， y=0， 数值显示为 0.00，表示第 0 个 epoch 时，权值为 0.306 的个数为 0.00。

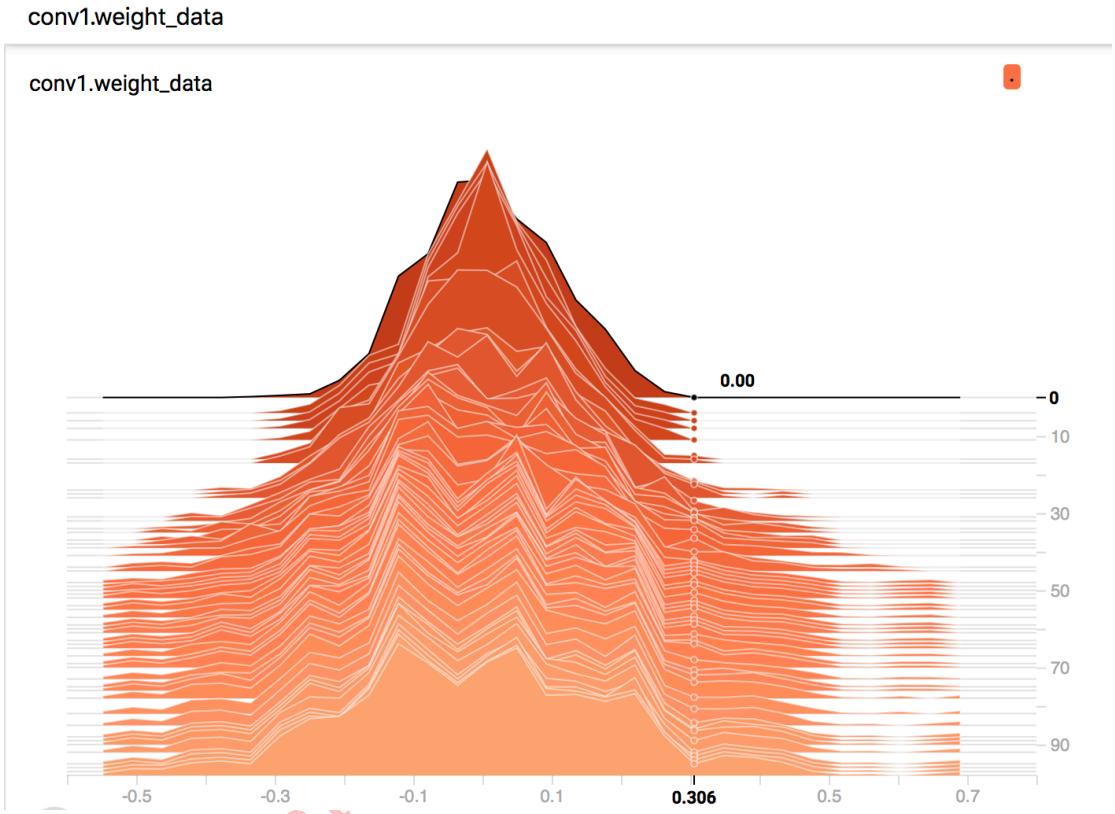
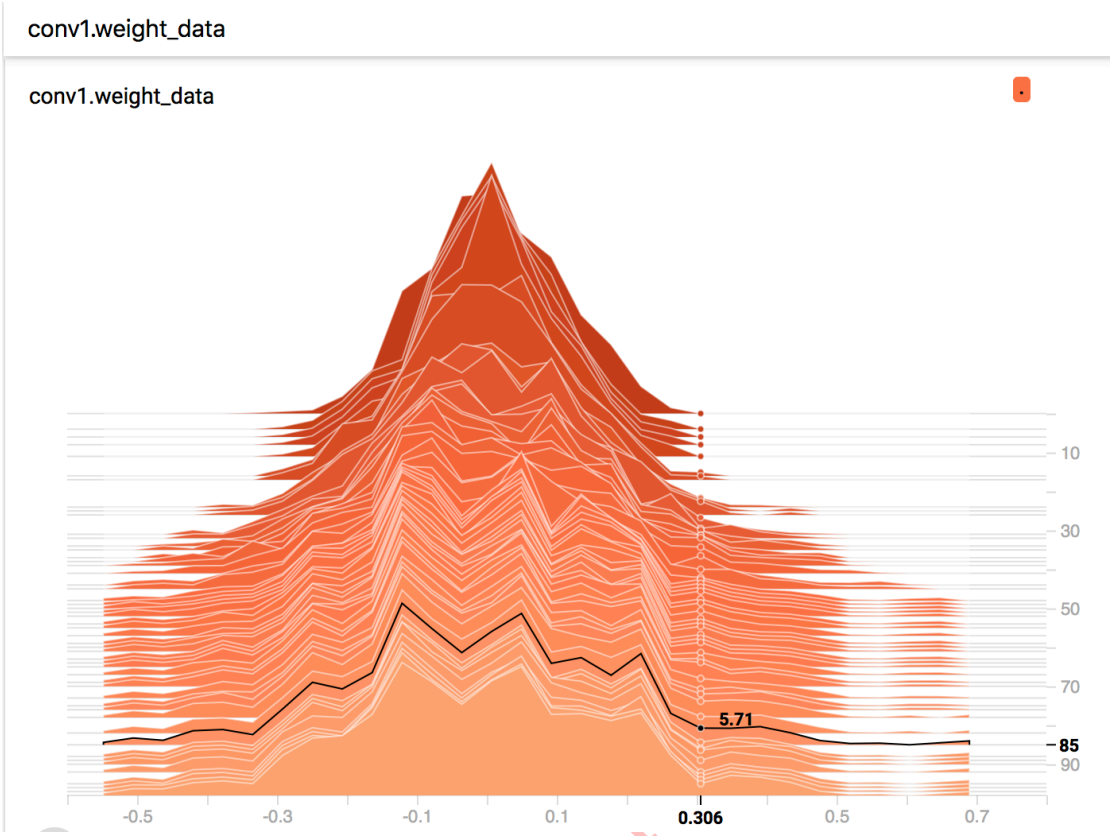


图 2， x=0.306， y=85， 数值显示为 5.71，表示第 85 个 epoch 时，权值在 0.306 区间的有 5.71 个。这里暂时还不明白为什么会是小数，知道的朋友可以补充一下。



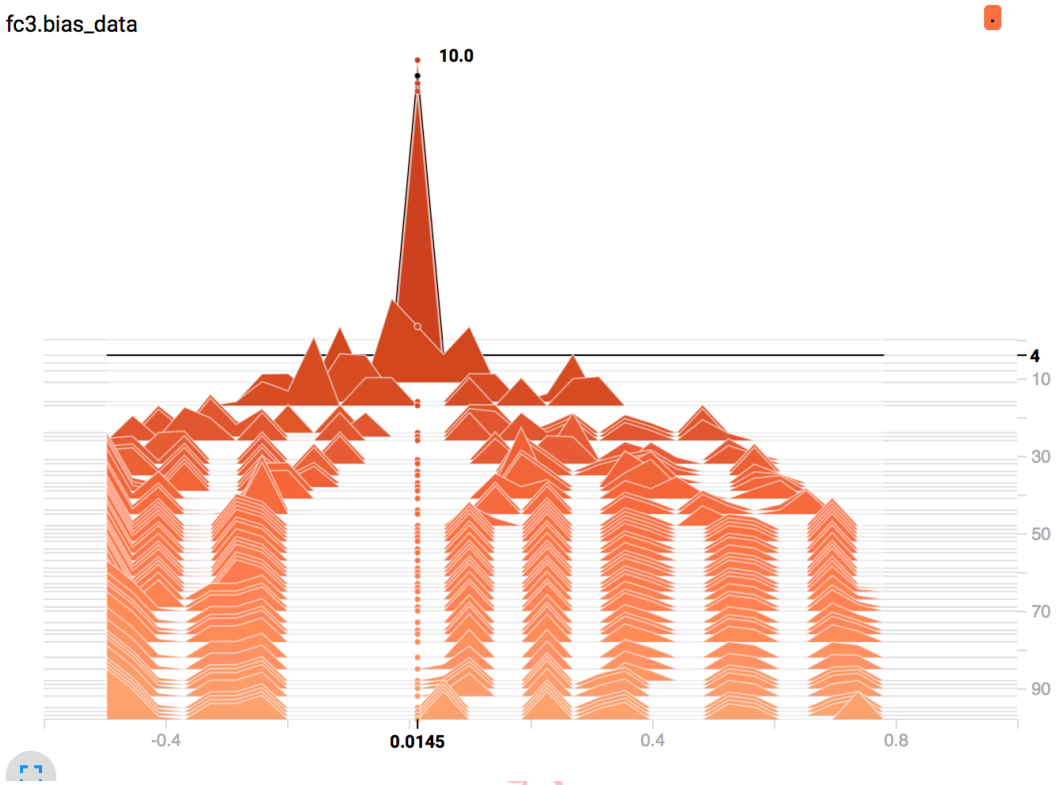
通过 HISTOGRAMS 可以看到第一个卷积层的权值随着训练的不断的“扩散”，扩大，一开始是个比较标准的高斯分布，并且最大值不会超过 0.3。

而到了后期，权值会发散到 0.6+，这个问题也是需要关注的，若权值太大容易导致过拟合。因为模型的输出值会被该特征所主导，从而引起过拟合现象，这个可以通过权值衰减(weight_decay)来缓解。

2. 偏置 bias 的监控

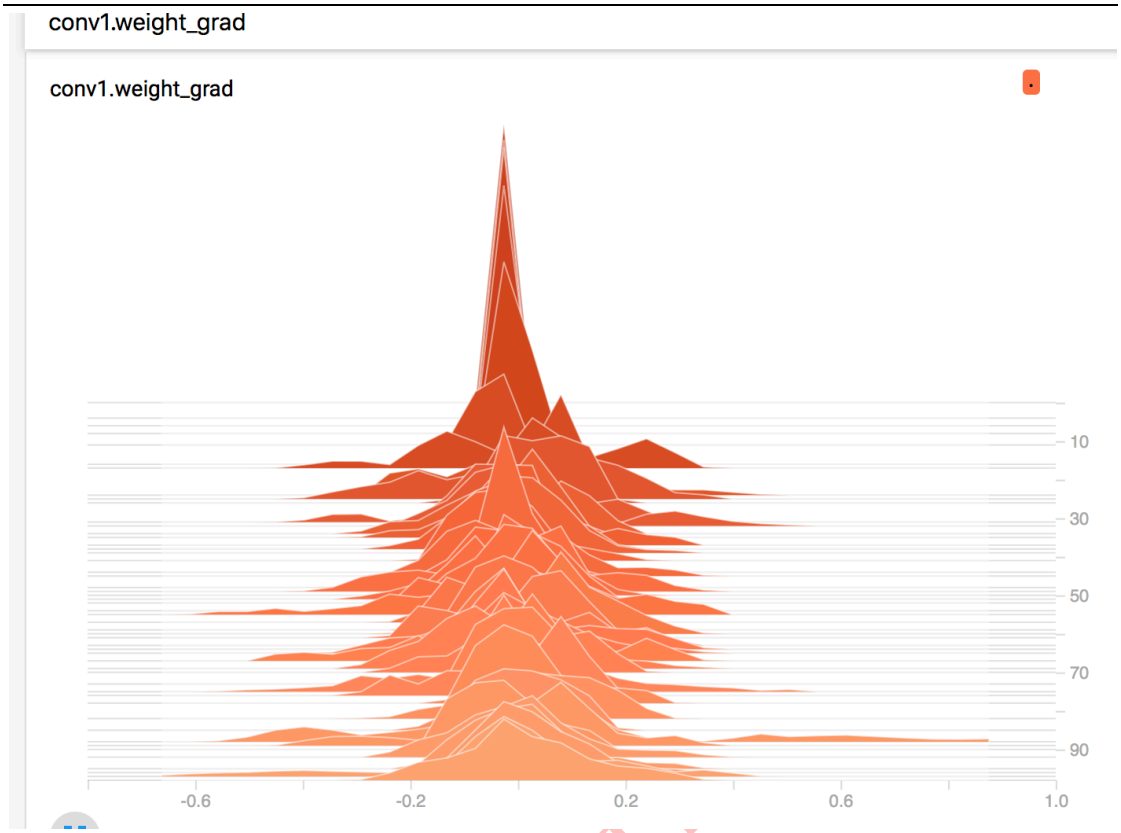
通常会监控输出层的 bias 的大小，若有特别大，或者特别小的 bias，那么某一类别的召回率可能会很低，可以通过观察输出层的 bias 来诊断是否在这一环节出问题。

从图上可以看到，一开始 10 个类别的 bias 都比较小，随着训练的进行，每个类别都有了自己的固定的 bias 大小。



3. 梯度的监控

下图为第一个卷积层权值的梯度变化情况，可以看到，几乎都是服从高斯分布的。倘若前面几层的梯度非常小，那么就是梯度流通不畅导致的，可以考虑残差结构或者辅助损失层等 trick 来解决梯度消失。



文末思考：

1. 通过观察各层的梯度，权值分布，我们可以针对性的设置学习率，为那些梯度小的层设置更大的学习率，让那些层可以有效的更新，不知道这样是否有用，大家可以尝试。

2. 对权值特别大的那些层，可以考虑为那一层设置更大的 weight_decay，是否能有效降低该层权值大小呢。

3. 通过对梯度的观察，可以合理的设置梯度 clip 的值喔。

4.5 混淆矩阵及其可视化

在分类任务中，个人十分喜欢混淆矩阵，通过混淆矩阵可以看出模型的偏好，而且对每一个类别的分类情况都了如指掌，为模型的优化提供很大帮助。本小节就介绍混淆矩阵概念及其可视化。

1. 混淆矩阵概念

混淆矩阵(Confusion Matrix)常用来观察分类结果，其是一个 N*N 的方阵，N 表示类别数。混淆矩阵的行表示真实类别，列表示预测类别。例如，猫狗的二分类问题，有猫的图像 10 张，狗的图像 30 张，模型对这 40 张图片进行预测，得到的混淆矩阵为

	阿猫	阿狗
阿猫	7	3
阿狗	10	20

从第一行中可知道，10 张猫的图像中，7 张预测为猫，3 张预测为狗，猫的召回率(Recall)为 7/10 = 70%，

从第二行中可知道，30 张狗的图像中，8 张预测为猫，22 张预测为狗，狗的召回率为 20/30 = 66.7%，

从第一列中可知道，预测为猫的 17 张图像中，有 7 张是真正的猫，猫的精确度(Precision)为 7 / 17 = 41.17%

从第二列中可知道，预测为狗的 23 张图像中，有 20 张是真正的狗，狗的精确度(Precision)为 20 / 23 = 86.96%

模型的准确率(Accuracy)为 7+20 / 40 = 67.5%

可以发现通过混淆矩阵可以清晰的看出网络模型的分类情况，若再结合上颜色可视化，可方便的看出模型的分类偏好。

本小节将介绍，混淆矩阵的统计及其可视化。

2. 混淆矩阵的统计

第一步：创建混淆矩阵

获取类别数，创建 N*N 的零矩阵

```
conf_mat = np.zeros([cls_num, cls_num])
```

第二步：获取真实标签和预测标签

labels 为真实标签，通常为一个 batch 的标签

predicted 为预测类别，与 labels 同长度

第三步：依据标签为混淆矩阵计数

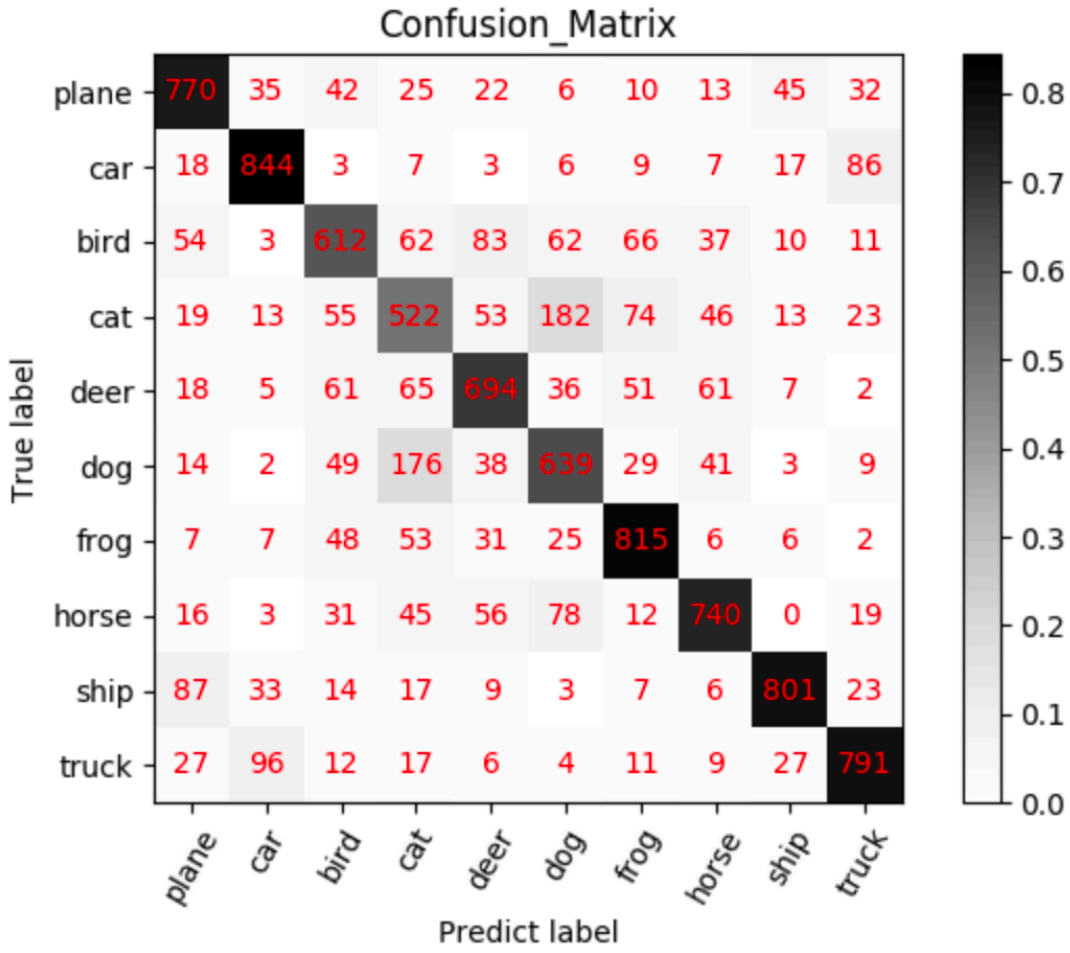
```
for i in range(len(labels)):
true_i = np.array(labels[i])
pre_i = np.array(predicted[i])
conf_mat[true_i, pre_i] += 1.0
```

3. 混淆矩阵可视化

可视化还需要各类标签名，存储为一个 list，以 cifar10 为例：

```
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
```

可视化效果如下图：



函数位于：

/Code/4_viewer/5_Show_ConfMat.py

运行/Code/main_training/main.py 即可在 /Result/your_time/文件夹下面看到混淆矩阵图啦。

本教程是以实际应用、工程开发为目的，着重介绍模型训练过程中遇到的实际问题，实际方法。如下图所示，在机器学习模型开发中，主要涉及三大部分，分别是数据、模型和损失函数及优化器。

通过本教程，希望能够给大家带来一个清晰的模型训练结构。当模型训练遇到问题时，首先通过可视化工具对数据、模型、损失等内容进行观察，分析并定位问题出在数据部分？模型部分？还是优化器？只有这样不断的通过可视化诊断你的模型，不断的对症下药，才能训练出一个较满意的模型。

结束语：

内容到此结束，希望本教程能给大家带来帮助！由于本人也是 PyTorch 的初学者，若在文中有任何错误，请大家多多指教，我将即时更改，以避免误导大家。

本人目前仍在学习 PyTorch，也在计划写更多关于 PyTorch 和机器学习的内容，若有需要的朋友或组织，请联系我！