Lab 9: Image Classification 实验报告

侯新铭 2021201651

目录：

本次 Lab 的目标是学习经典的 CV 网络结构及其变体，采用经典的 CIFAR-10 Dataset 数据集；之前学完 python 课后的暑假自学计算机视觉时，我接触过这个数据集，学习了最基本的 MLP 和 CNN 两个模型，就没有继续学。借助本次 lab，我进一步查阅资料学习了上课老师讲的 ResNet 的更加细节的原理和思路，相关探究和心得体会已写在了此报告中；代码层面，我探索了从数据的加载、基本的库的使用、到整体 ResNet 模型的实现；此外，我探究了训练过程中可以采用的三种效果还不错的优化方式：

1. 对于学习率，采用“一周期内学习率改变法则”
2. 正则化
3. 梯度剪切

最后，我将 ResNet 和 MLP,CNN 两个模型的运行效果进行了可视化对比。

一、 数据理解与数据加载

1. 数据理解

CIFAR-10 数据集的内容：
10 个不同类别
每个类别 6000 张图像，共计 60000 张
图像分辨率为 32x32，均为 RGB 三通道图像
如右侧 Figure 1 **示意所示**

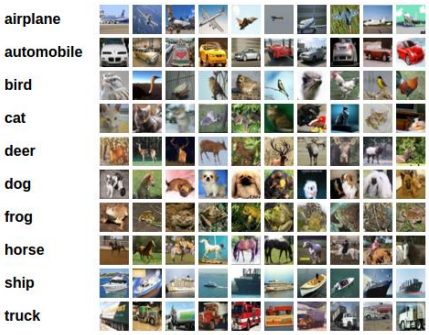


Figure 1

2. 数据载入

在搜索 CIFAR-10 的过程中，我发现其作为经典的用于训练机器学习和计算机视觉算法的数据集，存在相应封装好的 torchvision 库 ，于是，我在 pytorch 官网（ https://pytorch.org/vision/stable/index.html ）进一步了解了这个库集成了的数据集，模型架构，以及常用图像处理操作等。

- torchvision.datasets
  - CelebA
  - CIFAR
  - Cityscapes
  - COCO
  - DatasetFolder
  - EMNIST
  - FakeData
  - Fashion-MNIST
  - Flickr
  - HMDB51
  - ImageFolder
  - ImageNet
  - Kinetics-400
  - KMNIST
- torchvision.models
  - Classification
  - Semantic Segmentation
  - Object Detection, Instance Segmentation and Person Keypoint Detection
  - Video classification
- torchvision.transforms
  - Scriptable transforms
  - Compositions of transforms
  - Transforms on PIL Image and torch.*Tensor
  - Transforms on PIL Image only
  - Transforms on torch.*Tensor only
  - Conversion Transforms
  - Generic Transforms
  - Functional Transforms
- torchvision.io
  - Video
  - Fine-grained video API
  - Image

本部分主要着眼于 torchvision.datasets，其中的基本特征可概括如下：
①所有数据集类都是 torch.utils.data,Dataset 的子类，它们实现了 __getitem__ 和 __len__ 方法。（正因都是 torch.utils.data,Dataset 的子类，都可以传递给一个 torch.utils.data.DataLoaderder，进而可以使用 torch.multiprocessing workers 并行加载多个样本！）
②并且，它们都有几乎相似的 API，都有两个共同的参数: transform 和 target _ transform，分别用于转换输入和目标。
下面是我们需要的 CIFAR10 类的介绍：

CIFAR10

CLASS torchvision.datasets.CIFAR10(*root: str, train: bool = True, transform: Optional[Callable] = None, target_transform: Optional[Callable] = None, download: bool = False*) [SOURCE]

CIFAR10 Dataset.

Parameters

- **root** (*string*) – Root directory of dataset where directory `cifar-10-batches-py` exists or will be saved to if download is set to True.
- **train** (*bool, optional*) – If True, creates dataset from training set, otherwise creates from test set.
- **transform** (*callable, optional*) – A function/transform that takes in an PIL image and returns a transformed version. E.g, `transforms.RandomCrop`
- **target_transform** (*callable, optional*) – A function/transform that takes in the target and transforms it.
- **download** (*bool, optional*) – If true, downloads the dataset from the internet and puts it in root directory. If dataset is already downloaded, it is not downloaded again.

由此，可直接基于 torchvision 库对 CIFAR-10 数据加载

首先 Import

```
from torchvision.datasets import CIFAR10
```

```
import torchvision.transforms as transforms
```

注意到前面所述的 CIFAR-10 类中有待传入参数 transform,
我们首先需要实例化我们想要的 transform 以对图片数据作预处理
1. 处理 RGB 像素值，需要 ToTensor()来把图片的灰度范围从 0~255 变成 0~1 之间
2. ToTensor 后，为加快后续学习过程中模型的收敛，还需要 transforms.Normalize()来对图像进行标准化 目标是：均值变为 0，标准差变为 1
转化公式为：

$$output = (input - mean) / std$$

其中，mean:各通道的均值 std：各通道的标准差；对于图片 RGB 三个通道，前一个(0.5,0.5,0.5)是设置各通道分别的 mean 值 后一个(0.5,0.5,0.5)是设置各通道分别的标准差
3. 最后，通过 transforms.Compose 将上述多个步骤组合在一起

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

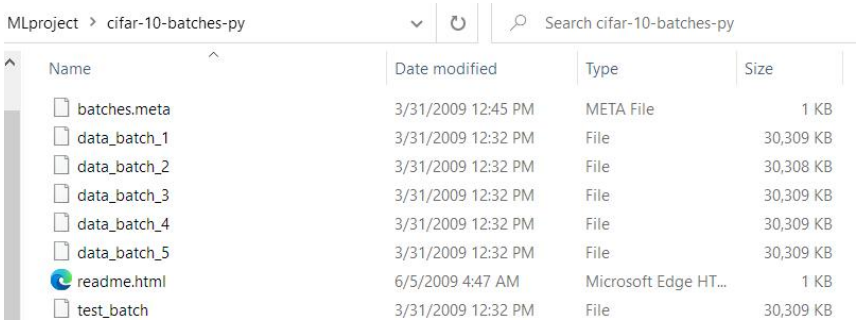
做好了上述准备，我们将 CIFAR-10 数据集下载并实例化为训练集和测试集两个对象

```
train_dataset = CIFAR10(root='.',
                        train=True,
                        transform=transform,
                        download=True)

test_dataset  = CIFAR10(root='.',
                        train=False,
                        transform=transform,
                        download=True)

print(type(train_dataset),type(test_dataset))
```

```
Files already downloaded and verified
Files already downloaded and verified
<class 'torchvision.datasets.cifar.CIFAR10'> <class 'torchvision.datasets.cifar.CIFAR10'>
```

- **train** (*bool, optional*) – If True, creates dataset from training set, otherwise creates from test set.

下载后的文件呈现



不妨直接读取下载好的 files 来操作

字典 dict 类型，方便后续操作实际上，在前面的 CIFAR-10 实例化为两个对象的过程中，各个文件中的数据已经以 tensor 的形式放在了这两个对象内；

因为对字典的操作是更简便的，我们不妨利用 read_pickle 把下载好的文件分别直接读取为 dict 字典类型（我们以 batch_1 为例来看各个 batch）

pandas.read_pickle

pandas.read_pickle(*filepath_or_buffer*, *compression='infer'*, *storage_options=None*) [source]

compression : *str or dict, default 'infer'*

For on-the-fly decompression of on-disk data. If 'infer' and 'filepath_or_buffer' is path-like, then detect compression from the following extensions: '.gz', '.bz2', '.zip', '.xz', or '.zst' (otherwise no compression). If using 'zip', the ZIP file must contain only one data file to be read in. Set to `None` for no decompression. Can also be a dict with key `'method'` set to one of {`'zip'`, `'gzip'`, `'bz2'`, `'zstd'`} and other key-value pairs are forwarded to `zipfile.ZipFile`, `gzip.GzipFile`, `bz2.BZ2File`, or `zstandard.ZstdDecompressor`, respectively. As an example, the following could be passed for Zstandard decompression using a custom compression dictionary: `compression={'method': 'zstd', 'dict_data': my_compression_dict}`.

Changed in version 1.4.0: Zstandard support.

storage_options : *dict, optional*

Extra options that make sense for a particular storage connection, e.g. host, port, username, password, etc. For HTTP(S) URLs the key-value pairs are forwarded to `urllib` as header options. For other URLs (e.g. starting with "s3://", and "gcs://") the key-value pairs are forwarded to `fsspec`. Please see `fsspec` and `urllib` for more details.

```
data_batch_1 = pd.read_pickle(r'./cifar-10-batches-py/data_batch_1')
data_meta    = pd.read_pickle(r'./cifar-10-batches-py/batches.meta')
```

130 ms [19]

```
data_batch_1 = pd.read_pickle(r'./cifar-10-batches-py/data_batch_1')
data_meta    = pd.read_pickle(r'./cifar-10-batches-py/batches.meta')
print(type(data_batch_1),type(data_meta))
print("==============")

print("data_batch_1 keys:")
print(data_batch_1.keys())
print("==============")

print("data_meta:")
print(data_meta.keys())
print(data_meta.values())
```

```
<class 'dict'> <class 'dict'>
==============
data_batch_1 keys:
dict_keys(['batch_label', 'labels', 'data', 'filenames'])
==============
data_meta:
dict_keys(['num_cases_per_batch', 'label_names', 'num_vis'])
dict_values([10000, ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck'], 3072])
```

可见，batch_1 包含的键有['batch_label', 'labels', 'data', 'filenames']

经查询，values 形式概括为如下所述：

data	a 10000×3072 array(uint85)，array的每行是一张32×32的彩色图片，前1024是red channel的值，后面1024是green channel的值，最后1024是blue channel的值。图片是**以行主顺序存储**，所以，前数组中前32个数表示的是一张图片第一行的red channel values。
labels	a list of 10000 numbers in the range 0-9，labels中的序号i对应的是data数组中第i个图片的label。
batch_label	batch的名称
filenames	数据集中data对应的图片名称数组

3. CHECKCHECK

3.1 查看标签与标签名以及它们的对应关系：

```
87 ms
labels = set(data_batch_1['labels'])
labels
```

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

```
165 ms [13]
labels_name = data_meta['label_names']
labels_name
```

```
['airplane',
 'automobile',
 'bird',
 'cat',
 'deer',
 'dog',
 'frog',
 'horse',
 'ship',
 'truck']
```

```
92 ms [16]
label_dict = {k:v for k,v in zip(labels, labels_name)} # dict(zip(labels, one_hot))
label_dict
```

```
{0: 'airplane',
 1: 'automobile',
 2: 'bird',
 3: 'cat',
 4: 'deer',
 5: 'dog',
 6: 'frog',
 7: 'horse',
 8: 'ship',
 9: 'truck'}
```

One_Hot Encoding 编码模式

值得一提的是，查资料过程中我发现，实际上 labels 采取了 One_Hot Encoding 编码模式，机器学习任务中，特征 labels 并不总是连续值，很多时候是分类值。我们将其特征用数字表示，以提高表示与运算的效率。但是，转化为数字表示后，数据不能直接用在分类器中。因为，分类器往往默认数据数据是连续且有序的。但按上述表示的数字并不有序的，而是随机分配的。

解决方法即为 One_Hot Encoding 独热编码，其方法是使用 N 位状态寄存器来对 N 个状态进行编码，每个状态都有它独立的寄存器位，并且在任何时候，其中只有一位激活。

例如：

自然状态码为：000,001,010,011,100,101

独热编码为：000001,000010,000100,001000,010000,100000

可以理解为：对于每一个特征，如果它有 m 个可能值，那么经过独热编码后，就变成了 m 个二元特征。并且，这些特征互斥，每次只有一个激活。因此，数据得以变成稀疏的了！

这样做的好处主要有：

1 解决了分类器不好处理属性数据的问题

2 在一定程度上也起到了扩充特征的作用

3.2 查看实例化后的两个数据集类分别的数据量是多少

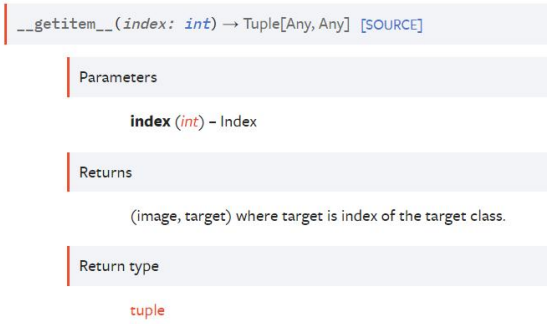
我们之前在 download CIFAR-10 的数据时，作为 CIFAR-10 类对象的测试集和训练集中到底有多少数据量是事先定好的，我们至此并不知道，于是我们欲查看一下，
这里借助了之前提到过的 CIFAR-10 类本身就已经实现了 __len__ 方法：

99 ms [27]
```
# 查看实例化后的两个数据集分别的数据量是多少
print(f'Train data shape: {len(train_dataset)}')
print(f'Test data shape:  {len(test_dataset)}')
```

```
Train data shape: 50000
Test data shape:  10000
```

3.3 在训练集维度（而非单单某个 batch）随机查看训练集类内的 case

这里需要借助之前提到过的 CIFAR-10 类本身就已经实现了 __getitem__ 方法：



__getitem__(*index: int*) → Tuple[Any, Any] [SOURCE]

Parameters

index (*int*) – Index

Returns

(image, target) where target is index of the target class.

Return type

tuple

代码如下：

165 ms [31]
```
# 随机检查训练集集数据中的图像形状及label
import random

x = random.randint(0,len(train_dataset)-1)
print("x=", x)
image, label = train_dataset[x]
print(type(image))
print(image.shape)
print(label)
```

```
x= 22877
torch.Size([3, 32, 32])
<class 'torch.Tensor'>
2
```

下面我们来作更有趣的可视化，看看训练数据集中的图片到底长什么样子，对应的标签是什么
但是，在实例化的过程中，我们自己定义 transform 方法传入给了 CIFAR10 类（回顾如下图）

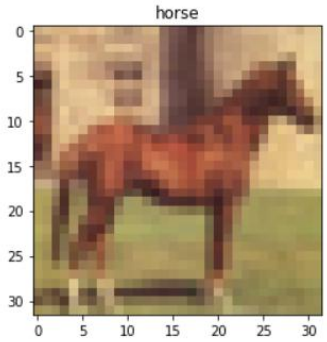
```
train_dataset = CIFAR10(root='.',
                        train=True,
                        transform=transform,
                        download=True)
```

因此，想要看到原本的图片样子，我们需要把 transform 中所作的标准化过程还原
（而 ToTensor 是不用的，plt 是可以正常显示它的）
采用除 2+0.5 的方式，形成[-1,1]→[0,1]的线性映射
另外，要通过 permute 方法来调整 img 的参数顺序（由（channels,imagesize,imagesize）转化为（imagesize,imagesize,channels）后方可显示）使其与 plt 的接口正确：

193 ms [43]
```python
# 我们想随机检查一下训练集中的图片的样子及与label的对应性

# 自定义imshow函数
def imshow(img, label):
    img = img / 2 + 0.5   # unnormalize
    # npimg = img.numpy()
    # plt.imshow(np.transpose(npimg, (1, 2, 0))) # tensor.permute = np.transpose, 二者等价
    plt.imshow(img.permute(1,2,0))
    plt.title(''.join(f'{label_dict.get(label)}' )) # 显示label_name

x = random.randint(0,len(train_dataset)-1)
print("x=", x)
image, label = train_dataset[x]

imshow(image, label)

plt.show()
```

193 ms [43]
```python
# 我们想随机检查一下训练集中的图片的样子及与label的对应性

# 自定义imshow函数
def imshow(img, label):
    img = img / 2 + 0.5   # unnormalize
    # npimg = img.numpy()
    # plt.imshow(np.transpose(npimg, (1, 2, 0))) # tensor.permute = np.transpose, 二者等价
    plt.imshow(img.permute(1,2,0))
    plt.title(''.join(f'{label_dict.get(label)}' )) # 显示label_name

x = random.randint(0,len(train_dataset)-1)
print("x=", x)
image, label = train_dataset[x]

imshow(image, label)

plt.show()
```

```
x= 36394
```



在一开始介绍 torchvision.datasets 的时候，我们提到它是 torch.utils.data,Dataset 的子类，都可以传递给一个 torch.utils.data.DataLoaderder，进而可以使用 torch.multiprocessing workers 并行加载多个样本!

什么是 DataLoader 呢？重要吗？必要吗？下面我们来解决这些问题。

pytorch 官网上的描述如下：

These options are configured by the constructor arguments of a `DataLoader`, which has signature:

```
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, *, prefetch_factor=2,
           persistent_workers=False)
```

```
dataset：Dataset类型，从其中加载数据
batch_size：int，可选。每个batch加载多少样本
shuffle：bool，可选。为True时表示每个epoch都对数据进行洗牌
sampler：Sampler，可选。从数据集中采样样本的方法。
num_workers：int，可选。加载数据时使用多少子进程。默认值为0，表示在主进程中加载数据。
collate_fn：callable，可选。
pin_memory：bool，可选
drop_last：bool，可选。True表示如果最后剩下不完全的batch,丢弃。False表示不丢弃。
```

我们在此重点关注其中的 Batch_Size 参数：

是机器学习中一个重要参数，它表示一次性读入多少批量的图片。

Batch_size 的作用：决定了下降的方向。

在合理范围内，增大 Batch_size 的**好处**：

1. 提高了内存利用率以及大矩阵乘法的并行化效率;
2. 跑完一次 epoch(全数据集）所需要的迭代次数减少，对相同的数据量，处理的速度比小的 Batch_size 要更快;
3. 在一定范围内，一般来说 Batch_Size 越大，其确定的下降方向越准，引起训练震荡越小。

盲目增大 Batch_size，Batch_size 过大的**坏处**：

1. 提高了内存利用率，但是内存容量可能撑不住;
2. 跑完一次 epoch(全数据集)所需的迭代次数减少，要想达到相同的精度，其所花费的时间大大增加，从而对参数的修正也就显得更加缓慢;

一般而言，根据 GPU 显存，设置为最大，而且一般要求是 8 的倍数（比如 16，32，64），GPU 内部的并行计算效率最高。或者选择一部分数据，设置几个 8 的倍数的 Batch_Size，看看 loss 的下降情况，再选用效果更好的值。

总结为：

batch_size 设的大一些，收敛得快，也就是需要训练的次数少，准确率上升的也很稳定，但是实际使用起来精度不高;

batch_size 设的小一些，收敛得慢，可能准确率来回震荡，因此需要把基础学习速率降低一些，但是实际使用起来精度较高。

根据 Batch_size 的取值，分为：

Full Batch Learning：Batch_size=数据集大小，适用于小数据集。

Mini-batches Learning：Batch_size= N（自己设定），适用于大数据集。

Online Learning（在线学习）：Batch_size=1

```
[84] batch_size = 64          # hyper-parameter (超参数)
train_loader = torch.utils.data.DataLoader(
                              dataset = train_dataset,
                              batch_size = batch_size,
                              shuffle = True)

test_loader = torch.utils.data.DataLoader(
                              dataset = test_dataset,
                              batch_size = batch_size,
                              shuffle = True)
```

对 DataLoader 的 check：

```
dd = list(train_loader)
```

```
images, labels = dd[0]
print(images.shape)
print(labels.shape)
```

```
torch.Size([64, 3, 32, 32])
torch.Size([64])
```

二、 网络搭建

2.1 最简单最原汁原味的神经网络 MLP(Multi-Layer Perception)

我们基于生物神经元模型可得到多层感知器 MLP 的基本结构，最典型的 MLP 包括包括三层：输入层、隐层和输出层，MLP 神经网络不同层之间是全连接的（全连接的意思就是：上一层的任何一个神经元与下一层的所有神经元都有连接）。

```
# pytorch相关libraries 以及再命名
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
```

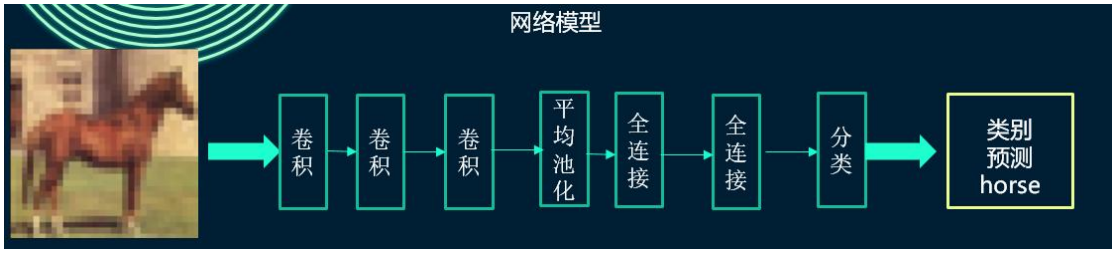
[87]
```
class MLP(torch.nn.Module):
  def __init__(self, input_size, n_hidden_units, n_classes):
    super(MLP, self).__init__()

    h1, h2, h3 = n_hidden_units

    # Add linear layers
    self.fc1 = nn.Linear(input_size,h1)
    self.fc2 = nn.Linear(h1, h2)
    self.fc3 = nn.Linear(h2, h3)
    self.fc4 = nn.Linear(h3, n_classes) # 10 classes
    self.dropout = nn.Dropout(0.25)

  def forward(self, x):
    x = x.view(-1, input_size) # Flatten out the input layer
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    x = F.relu(self.fc3(x))
    x = self.dropout(x)
    x = self.fc4(x)
    return x
```

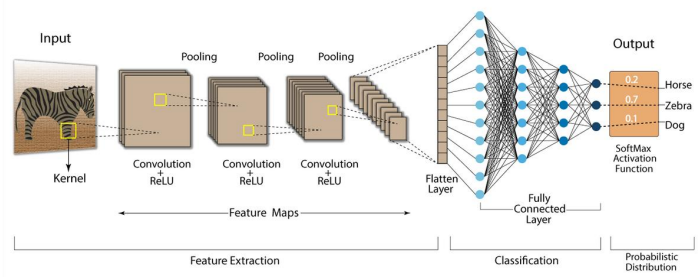
[88]
```
# 实例化 MLP model
input_size = IMAGE_HEIGHT*IMAGE_WIDTH*COLOR_CHANNELS
n_hidden_units = [512, 256, 128]

mlp_model = MLP(input_size, n_hidden_units, N_CLASSES)
```

[89]
```
# 检查输出的shape
output = mlp_model(images)
output.shape
```

```
torch.Size([64, 10])
```

2.2 卷积神经网络模型





[90]
```python
def conv3x3(in_channels, out_channels, stride=1):
  """3x3 convolution with padding"""
  return nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1, bias=False)
```

```python
class CNN(torch.nn.Module):
  def __init__(self, in_channels, n_classes):
    super(CNN, self).__init__()
    # Conv layers
    self.conv =conv3x3(in_channels=in_channels, out_channels=64)
    self.bn = nn.BatchNorm2d(64)
    self.relu = nn.ReLU(inplace=True)

    #self.conv0 =conv3x3(in_channels=in_channels, out_channels=64)
    #self.bn0 = nn.BatchNorm2d(64)

    # Residual block
    self.conv1 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=1, padding=1)
    self.bn1 = nn.BatchNorm2d(64)
    self.relu1 = nn.ReLU(inplace=True)

    self.conv2 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=1, padding=1)
    self.bn2 = nn.BatchNorm2d(64)
    self.relu2 = nn.ReLU(inplace=True)

    # Pooling
    self.avg_pool = nn.AvgPool2d(2)

    # Linear layers
    self.fc1 = nn.Linear(16384, 64 )
    self.fc2 = nn.Linear(64, n_classes)

  def forward(self, x):

    #print(identity.shape)
    out = self.conv(x)
    out = self.bn(out)
    out = self.relu(out)

    #out = self.conv0(x)
    #out = self.bn0(out)
    #out = self.relu(out)

    out = self.conv1(out)
    out = self.bn1(out)
    out = self.relu1(out)

    out = self.conv2(out)
    out = self.bn2(out)
    out = self.relu2(out)  # skip connection + out
    #print(out.shape)

    out = self.avg_pool(out)

    out = out.view(out.size(0), -1) # Flatten all dimension except batch
    #print(out.shape)

    out = self.fc1(out)
    out = F.relu(out)

    out = self.fc2(out)

    return out
```

```python
# 实例化 CNN model
cnn_model = CNN(in_channels=3, n_classes=10)
```

[93]
```python
images, labels = dd[0]
print(images.shape)
print(labels.shape)
```

```
torch.Size([64, 3, 32, 32])
torch.Size([64])
```

[94]
```python
# 检查输出的shape
output = cnn_model(images)
output.shape
```

```
torch.Size([64, 10])
```

2.3 RetNets

深度残差网络（Deep residual network, ResNet）的提出是 CNN 图像史上的一件里程碑事件 ResNets 要解决的是深度神经网络的“退化”问题(Degradation problem)。

什么是“退化”?

我们知道，对浅层网络逐渐叠加 layers，模型在训练集和测试集上的性能会变好，因为模型复杂度更高了，表达能力更强了，可以对潜在的映射关系拟合得更好。而“退化”指的是，给网络叠加更多的层后，性能却快速下降的情况。

训练集上的性能下降，可以排除过拟合，BN 层的引入也基本解决了 plain net 的梯度消失和梯度爆炸问题。如果不是过拟合以及梯度消失导致的，那原因是什么？

按道理，给网络叠加更多层，浅层网络的解空间是包含在深层网络的解空间中的，深层网络的解空间至少存在不差于浅层网络的解，因为只需将增加的层变成恒等映射，其他层的权重原封不动 copy 浅层网络，就可以获得与浅层网络同样的性能。更好的解明明存在，为什么找不到？找到的反而是更差的解？

显然，这是个优化问题，反映出结构相似的模型，其优化难度是不一样的，且难度的增长并不是线性的，越深的模型越难以优化。

有两种解决思路，

一种是调整求解方法，比如更好的初始化、更好的梯度下降算法等；

另一种是调整模型结构，让模型更易于优化。

ResNets 的作者从后者入手，探求更好的模型结构。将堆叠的几层 layer 称之为一个 block，对于某个 block，其可以拟合的函数为 F(x)，如果期望的潜在映射为 H(x)，与其让 F(x) 直接学习潜在的映射，不如去学习残差 H(x)−x，即 F(x):=H(x)−x，这样原本的前向路径上就变成了 F(x)+x，用 F(x)+x 来拟合 H(x)。作者认为这样可能更易于优化，因为相比于让 F(x)学习成恒等映射，让 F(x)学习成 0 要更加容易——后者通过 L2 正则就可以轻松实现。这样，对于冗余的 block，只需 F(x)→0 就可以得到恒等映射，性能不减。

```python
class BasicResidualBlock(nn.Module):
  def __init__(self):
    super(BasicResidualBlock, self).__init__()
    self.conv1 = nn.Conv2d(in_channels=3, out_channels=3, kernel_size=3, stride=1, padding=1)
    self.relu1 = nn.ReLU()
    self.conv2 = nn.Conv2d(in_channels=3, out_channels=3, kernel_size=3, stride=1, padding=1)
    self.relu2 = nn.ReLU()

  def forward(self, x):
    identity = x
    out = self.conv1(x)
    out = self.relu1(out)
    out = self.conv2(out)
    out = self.relu2(out)
    return self.relu2(out) + identity # ReLU被应用于加上identity/input的前和后
```

```
[96] simple_resblock = BasicResidualBlock()
     images, labels = dd[0]
     print(images.shape)
     print(labels.shape)

     torch.Size([64, 3, 32, 32])
     torch.Size([64])

[97] out = simple_resblock(images)
     out.shape

     torch.Size([64, 3, 32, 32])
```

```python
class ResNet9(nn.Module):
  def __init__(self, in_channels, n_classes):
    super(ResNet9, self).__init__()

    self.conv1 = conv_block(in_channels, 64)
    self.conv2 = conv_block(64, 128, pool=True)
    self.res1  = nn.Sequential(conv_block(128,128), conv_block(128,128))

    self.conv3 = conv_block(128, 256, pool=True)
    self.conv4 = conv_block(256, 512, pool=True)
    self.res2  = nn.Sequential(conv_block(512,512), conv_block(512,512))

    self.classifier = nn.Sequential(nn.MaxPool2d(4),
                                    nn.Flatten(),
                                    nn.Linear(512, 128),
                                    nn.Linear(128, n_classes))

  def forward(self, x):
    out = self.conv1(x)
    out = self.conv2(out)
    out = self.res1(out) + out # skip connection
    out = self.conv3(out)
    out = self.conv4(out)
    out = self.res2(out) + out # skip connection
    out = self.classifier(out)
    return out
```

```python
def conv_block(in_channels, out_channels, pool=False):
  layers = [nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True)]
  if pool:
    layers.append(nn.MaxPool2d(2))
  return nn.Sequential(*layers)
```

交叉熵

$$H(p,\ q) = -\sum_{y} p(y) \cdot \log q(y)$$

衡量真实分布$p(y)$和预测分布$q(y)$之间的差异

如果$p(y) = q(y)$，那么交叉熵$H(p,\ q)$最小，且刚好等于$p(y)$的熵

从二分类出发：$y \in \{0,\ 1\}$

$$H(p,\ q) = -\sum_{y} p(y) \cdot \log q(y)$$

$$= -\ y \cdot \log q(y=1) + (1-y) \cdot \log\big(1 - q(y=1)\big)$$

在所有数据上取平均： $-\frac{1}{N}\sum_{i=1}^{N}[y_i \ln q_i + (1-y_i)\ln(1-q_i)]$

逻辑回归： $q(y=1) = \frac{1}{1+e^{-(\boldsymbol{x}\boldsymbol{w}+b)}}$

推广到多分类： 假设总共有 C 类，$y \in \{0,\ 1,2,\ldots,C-1\}$

$$H(p,\ q) = -\sum_{y} p(y) \cdot \log q(y) = \sum_{y=0}^{C-1} p(y) \cdot \log q(y)$$

b. 这就产生了一个问题：如何得到$q(y)$?

模型输出一个 C 维向量$\boldsymbol{z} = [z[0],\ z[1],\ \ldots, z[C-1]] \in R^C$

利用 softmax 函数计算:

$$q(y=i) = \frac{e^{z[i]}}{\sum_j e^{z[j]}}$$

PyTorch 提供的交叉熵损失函数：nn.CrossEntropyLoss

输入：

NxC 维矩阵$\boldsymbol{Z}$，其中每一行为$\boldsymbol{z} = [z[0],\ z[1],\ \ldots, z[C-1]] \in R^C$

N 维向量$\boldsymbol{y}$，其中每个元素为$y \in \{0,\ 1,2,\ldots,C-1\}$

注意：该损失函数同时计算了上面我们提到的 softmax 函数和交叉熵函数：

$loss(\boldsymbol{Z}$

CLASS `torch.nn.CrossEntropyLoss(weight: Optional[torch.Tensor] = None, size_average=None, ignore_index: int = -100, reduce=None, reduction: str = 'mean')` [SOURCE]

$\boldsymbol{y}) = -$

$\sum_{i=0}^{N-1} \log\left(\frac{e^{\boldsymbol{Z}[i,\ y[i]]}}{\sum_j e^{\boldsymbol{Z}[i,\ j]}}\right)$

The loss can be described as: Softmax function

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

2.4 预测函数

```
def prediction(data_loader, model, criterion, cuda=None):
  correct = 0
  total = 0
  losses = 0

  for i, (images, labels) in enumerate(data_loader):
    if cuda is not None:
      # switch tensor type to GPU
      images = images.cuda()
      labels = labels.cuda()

    # Flatten the images
    #images = images.view(-1, input_size)

    outputs = model(images)

    loss = criterion(outputs, labels)

    _, predictions = torch.max(outputs, dim=1)

    correct += torch.sum(labels == predictions).item()
    total += labels.shape[0]

    losses += loss.data.item()

  return losses/len(list(data_loader)), 1 - correct/total
```

2.5 训练函数中引入一些优化过程

1. 对于学习率，采用“一周期内学习率改变法则”：

从较低的学习率开始，在大约 30%的 epoch 中，逐批增加到较高的学习率，然后在剩余的 epoch 中逐渐降低到非常低的值。

```
def fit_one_cycle(epochs, max_lr, model, train_loader, val_loader, criterion,
                  weight_decay=0, grad_clip=None, opt_func=torch.optim.SGD, cuda=None)
```

2. 正则化，通过在损失函数中增加一项来防止权重过大

```
optimizer = opt_func(model.parameters(), max_lr, weight_decay=weight_decay)

sched = torch.optim.lr_scheduler.OneCycleLR(optimizer, max_lr, epochs=epochs,
                                            steps_per_epoch=len(train_loader))
```

3. 梯度剪切:可以将梯度值限制在一个较小的范围内，防止因梯度值过大导致参数发生不必要的变化。

```
if grad_clip is not None:
  nn.utils.clip_grad_value_(model.parameters(), grad_clip)
```

```
  if epoch%1 == 0:
    print('Epoch: {}/{}, last_lr: {:.5f}, train_loss: {:.4f}, val_loss: {:.4f}, val_accuracy: {:.1f}%'
          .format(epoch+1, epochs, results['lrs'][-1], train_loss/n_iter, test_loss, (1-train_error_rate)*100))

return history
```

二、 模型训练与结果

训练平台为华为云平台：


```
epochs = 20
max_lr = 0.01
grad_clip = 0.1
weight_decay = 1e-4
opt_func = torch.optim.Adam
criterion = nn.CrossEntropyLoss()
```

1. MLP

```
[107] history_mlp = fit_one_cycle(epochs, max_lr, mlp_model, train_loader, test_loader, criterion, weight_decay, opt_func=opt_func, cuda=cuda  )

Epoch: 1/20, last_lr: 0.00104, train_loss: 1.7133, val_loss: 1.6122, val_accuracy: 39.1%
Epoch: 2/20, last_lr: 0.00280, train_loss: 1.6077, val_loss: 1.6305, val_accuracy: 43.4%
Epoch: 3/20, last_lr: 0.00520, train_loss: 1.7468, val_loss: 1.8660, val_accuracy: 38.2%
Epoch: 4/20, last_lr: 0.00760, train_loss: 1.9208, val_loss: 2.1061, val_accuracy: 30.7%
Epoch: 5/20, last_lr: 0.00936, train_loss: 1.9791, val_loss: 1.9542, val_accuracy: 27.5%
Epoch: 6/20, last_lr: 0.01000, train_loss: 2.0736, val_loss: 2.1174, val_accuracy: 21.9%
Epoch: 7/20, last_lr: 0.00987, train_loss: 2.1655, val_loss: 2.3133, val_accuracy: 17.9%
Epoch: 8/20, last_lr: 0.00950, train_loss: 2.2127, val_loss: 2.1743, val_accuracy: 14.4%
Epoch: 9/20, last_lr: 0.00891, train_loss: 2.1893, val_loss: 2.1759, val_accuracy: 14.7%
Epoch: 10/20, last_lr: 0.00812, train_loss: 2.2362, val_loss: 2.2162, val_accuracy: 14.0%
Epoch: 11/20, last_lr: 0.00717, train_loss: 2.2375, val_loss: 2.2736, val_accuracy: 14.5%
Epoch: 12/20, last_lr: 0.00611, train_loss: 2.2495, val_loss: 2.2393, val_accuracy: 13.9%
Epoch: 13/20, last_lr: 0.00500, train_loss: 2.2448, val_loss: 2.1821, val_accuracy: 13.6%
Epoch: 14/20, last_lr: 0.00389, train_loss: 2.2209, val_loss: 2.2111, val_accuracy: 15.1%
Epoch: 15/20, last_lr: 0.00283, train_loss: 2.1917, val_loss: 2.1731, val_accuracy: 15.9%
Epoch: 16/20, last_lr: 0.00188, train_loss: 2.1740, val_loss: 2.1416, val_accuracy: 16.3%
Epoch: 17/20, last_lr: 0.00109, train_loss: 2.0587, val_loss: 1.9502, val_accuracy: 21.2%
Epoch: 18/20, last_lr: 0.00050, train_loss: 1.8688, val_loss: 1.7953, val_accuracy: 29.8%
Epoch: 19/20, last_lr: 0.00013, train_loss: 1.7465, val_loss: 1.7149, val_accuracy: 35.8%
Epoch: 20/20, last_lr: 0.00000, train_loss: 1.6739, val_loss: 1.6879, val_accuracy: 38.9%
```

2. CNN

```
[109] history_cnn = fit_one_cycle(epochs, max_lr, cnn_model, train_loader, test_loader, criterion, weight_decay, opt_func=opt_func, cuda=cuda )

Epoch: 1/20, last_lr: 0.00104, train_loss: 1.2494, val_loss: 1.0236, val_accuracy: 55.3%
Epoch: 2/20, last_lr: 0.00280, train_loss: 0.9631, val_loss: 0.9174, val_accuracy: 66.2%
Epoch: 3/20, last_lr: 0.00520, train_loss: 0.8766, val_loss: 0.9028, val_accuracy: 69.5%
Epoch: 4/20, last_lr: 0.00760, train_loss: 0.8171, val_loss: 0.8931, val_accuracy: 71.8%
Epoch: 5/20, last_lr: 0.00936, train_loss: 0.7908, val_loss: 0.9100, val_accuracy: 72.5%
Epoch: 6/20, last_lr: 0.01000, train_loss: 0.7709, val_loss: 0.8913, val_accuracy: 73.6%
Epoch: 7/20, last_lr: 0.00987, train_loss: 0.7529, val_loss: 0.9392, val_accuracy: 74.1%
Epoch: 8/20, last_lr: 0.00950, train_loss: 0.7363, val_loss: 0.8710, val_accuracy: 74.8%
Epoch: 9/20, last_lr: 0.00891, train_loss: 0.7136, val_loss: 0.9093, val_accuracy: 75.4%
Epoch: 10/20, last_lr: 0.00812, train_loss: 0.6784, val_loss: 0.8166, val_accuracy: 76.5%
Epoch: 11/20, last_lr: 0.00717, train_loss: 0.6265, val_loss: 0.8479, val_accuracy: 78.2%
Epoch: 12/20, last_lr: 0.00611, train_loss: 0.5694, val_loss: 0.8361, val_accuracy: 80.5%
Epoch: 13/20, last_lr: 0.00500, train_loss: 0.4862, val_loss: 0.7928, val_accuracy: 83.1%
Epoch: 14/20, last_lr: 0.00389, train_loss: 0.3963, val_loss: 0.8488, val_accuracy: 86.2%
Epoch: 15/20, last_lr: 0.00283, train_loss: 0.2912, val_loss: 0.8257, val_accuracy: 89.9%
Epoch: 16/20, last_lr: 0.00188, train_loss: 0.1804, val_loss: 0.8912, val_accuracy: 94.0%
Epoch: 17/20, last_lr: 0.00109, train_loss: 0.0972, val_loss: 0.9831, val_accuracy: 97.0%
Epoch: 18/20, last_lr: 0.00050, train_loss: 0.0451, val_loss: 1.0498, val_accuracy: 98.9%
Epoch: 19/20, last_lr: 0.00013, train_loss: 0.0242, val_loss: 1.0818, val_accuracy: 99.5%
Epoch: 20/20, last_lr: 0.00000, train_loss: 0.0179, val_loss: 1.0898, val_accuracy: 99.8%
```

```
PATH_CNN = './cnn_cifar_net.pth'
torch.save(cnn_model.state_dict(), PATH_CNN)
```

3. ResNets

```
18,8 min [111] history_resnet = fit_one_cycle(epochs, max_lr, resnet_model, train_loader, test_loader, criterion, weight_decay, opt_func=opt_func, cuda=cuda )
```

```
Epoch: 1/20, last_lr: 0.00104, train_loss: 1.0980, val_loss: 0.8677, val_accuracy: 60.9%
Epoch: 2/20, last_lr: 0.00280, train_loss: 0.7461, val_loss: 0.8171, val_accuracy: 74.4%
Epoch: 3/20, last_lr: 0.00520, train_loss: 0.7455, val_loss: 0.7513, val_accuracy: 75.1%
Epoch: 4/20, last_lr: 0.00760, train_loss: 0.7208, val_loss: 0.6921, val_accuracy: 75.8%
Epoch: 5/20, last_lr: 0.00936, train_loss: 0.6765, val_loss: 0.7439, val_accuracy: 76.9%
Epoch: 6/20, last_lr: 0.01000, train_loss: 0.6834, val_loss: 0.7072, val_accuracy: 76.6%
Epoch: 7/20, last_lr: 0.00987, train_loss: 0.6696, val_loss: 0.7575, val_accuracy: 77.2%
Epoch: 8/20, last_lr: 0.00950, train_loss: 0.6460, val_loss: 0.6679, val_accuracy: 78.1%
Epoch: 9/20, last_lr: 0.00891, train_loss: 0.6131, val_loss: 0.6175, val_accuracy: 79.0%
Epoch: 10/20, last_lr: 0.00812, train_loss: 0.5710, val_loss: 0.6628, val_accuracy: 80.5%
Epoch: 11/20, last_lr: 0.00717, train_loss: 0.5235, val_loss: 0.6024, val_accuracy: 82.0%
Epoch: 12/20, last_lr: 0.00611, train_loss: 0.4723, val_loss: 0.5332, val_accuracy: 83.9%
Epoch: 13/20, last_lr: 0.00500, train_loss: 0.4180, val_loss: 0.5400, val_accuracy: 85.6%
Epoch: 14/20, last_lr: 0.00389, train_loss: 0.3484, val_loss: 0.5172, val_accuracy: 88.0%
Epoch: 15/20, last_lr: 0.00283, train_loss: 0.2771, val_loss: 0.4972, val_accuracy: 90.3%
Epoch: 16/20, last_lr: 0.00188, train_loss: 0.1949, val_loss: 0.5234, val_accuracy: 93.2%
Epoch: 17/20, last_lr: 0.00109, train_loss: 0.1120, val_loss: 0.5367, val_accuracy: 96.2%
Epoch: 18/20, last_lr: 0.00050, train_loss: 0.0484, val_loss: 0.5535, val_accuracy: 98.7%
Epoch: 19/20, last_lr: 0.00013, train_loss: 0.0219, val_loss: 0.5726, val_accuracy: 99.6%
Epoch: 20/20, last_lr: 0.00000, train_loss: 0.0143, val_loss: 0.5828, val_accuracy: 99.9%
```

```
PATH_RESNET = './rescnn_cifar_lr_net.pth'
torch.save(resnet_model.state_dict(), PATH_RESNET)
```

三、 通过可视化进行模型评估

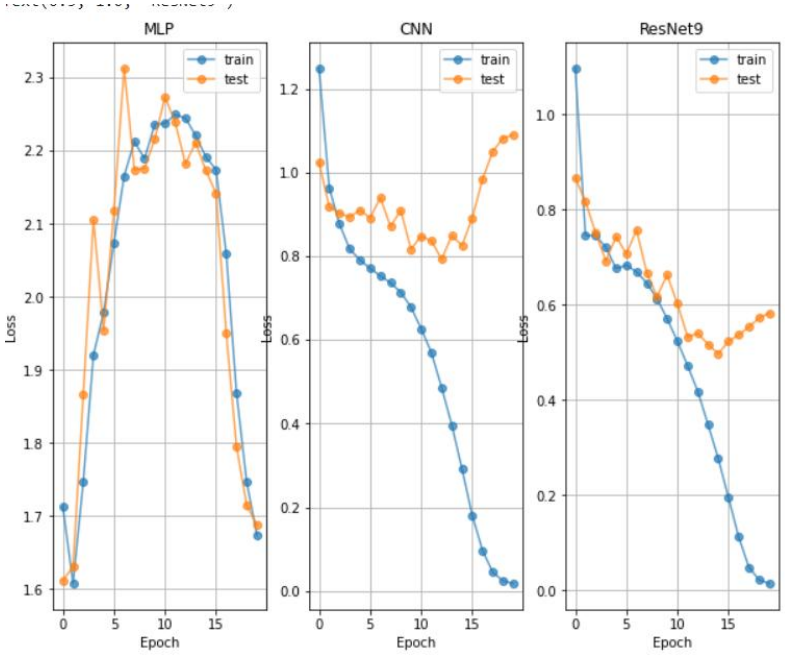
①定义了两个函数，初始化定义图像各个指标

```
40 ms [113] def plot_accuracies(history):
                accuracies = [x['val_acc'] for x in history]

                plt.plot(accuracies, '-x')
                plt.xlabel('epoch')
                plt.ylabel('accuracy')
                plt.title('Accuracy vs. No. of epochs');
```

```
18,8 min [114] def plot_losses(history):
                 #plt.figure(figsize=(10, 8))
                 #plt.subplot(1,2,1)
                 train_losses = [x.get('train_loss') for x in history]
                 test_losses = [x['val_loss'] for x in history]
                 plt.plot(train_losses, label='train', marker='o', alpha=0.7)
                 plt.plot(test_losses, label='test', marker='o', alpha=0.7)
                 plt.xlabel("Epoch")
                 plt.ylabel("Loss")
                 plt.grid(True)
                 plt.legend()
                 plt.title('Model Loss')
```

②训练和 test 过程中保存的结果中 loss 的可视化结果

```
[115] plt.figure(figsize=(10, 8))
      plt.subplot(1,3,1)
      plot_losses(history_mlp)
      plt.title('MLP')
      plt.subplot(1,3,2)
      plot_losses(history_cnn)
      plt.title('CNN')
      plt.subplot(1,3,3)
      plot_losses(history_resnet)
      plt.title('ResNet9')
```



③训练和 test 过程中保存的结果中 accuracy 的可视化结果

```
plt.figure(figsize=(10, 8))
plt.subplot(1,3,1)
plot_accuracies(history_mlp)
plt.title('MLP')
plt.subplot(1,3,2)
plot_accuracies(history_cnn)
plt.title('CNN')
plt.subplot(1,3,3)
plot_accuracies(history_resnet)
plt.title('ResNet9')
```