Kaggle上的图像分类（CIFAR-10)

现在，我们将运用在前面几节中学到的知识来参加Kaggle竞赛，该竞赛解决了CIFAR-10图像分类问题。比赛网址是https://www.kaggle.com/c/cifar-10

In [1]:

```
# 本节的网络需要较长的训练时间
# 可以在Kaggle访问:
# https://www.kaggle.com/boyuai/boyu-d2l-image-classification-cifar-10
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
import os
import time
```

In [2]:

```
print("PyTorch Version: ",torch.__version__)
```

```
PyTorch Version:  1.3.0
```

获取和组织数据集

比赛数据分为训练集和测试集。训练集包含 50,000 图片。测试集包含 300,000 图片。两个数据集中的图像格式均为PNG，高度和宽度均为32像素，并具有三个颜色通道（RGB）。图像涵盖10个类别：飞机，汽车，鸟类，猫，鹿，狗，青蛙，马，船和卡车。 为了更容易上手，我们提供了上述数据集的小样本。“ train_tiny.zip”包含 80 训练样本，而“ test_tiny.zip”包含100个测试样本。它们的未压缩文件夹名称分别是“ train_tiny”和“ test_tiny”。

图像增强

In [3]:

```
data_transform = transforms.Compose([
    transforms.Resize(40),
    transforms.RandomHorizontalFlip(),
    transforms.RandomCrop(32),
    transforms.ToTensor()
])
trainset = torchvision.datasets.ImageFolder(root='/home/kesci/input/CIFAR101419/cifar-10/cifar-10/train'
                                            , transform=data_transform)
```

In [4]:

```
trainset[0][0].shape
```

Out[4]:

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

In [5]:

```
data = [d[0].data.cpu().numpy() for d in trainset]
np.mean(data)
```

Out[5]:

```
0.46756765
```

In [6]:

```
np.std(data)
```

Out[6]:

```
0.23922285
```

In [7]:

```
# 图像增强
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),  #先四周填充0，再把图像随机裁剪成32*32
    transforms.RandomHorizontalFlip(),  #图像一半的概率翻转，一半的概率不翻转
    transforms.ToTensor(),
    transforms.Normalize((0.4731, 0.4822, 0.4465), (0.2212, 0.1994, 0.2010)), #R,G,B每层的归一化用到的均值和方差
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4731, 0.4822, 0.4465), (0.2212, 0.1994, 0.2010)),
])
```

导入数据集

In [8]:

```
train_dir = '/home/kesci/input/CIFAR101419/cifar-10/cifar-10/train'
test_dir = '/home/kesci/input/CIFAR101419/cifar-10/cifar-10/test'

trainset = torchvision.datasets.ImageFolder(root=train_dir, transform=transform_train)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=256, shuffle=True)

testset = torchvision.datasets.ImageFolder(root=test_dir, transform=transform_test)
testloader = torch.utils.data.DataLoader(testset, batch_size=256, shuffle=False)

classes = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'forg', 'horse', 'ship', 'truck']
```

定义模型

ResNet-18网络结构：ResNet全名Residual Network残差网络。Kaiming He 的《Deep Residual Learning for Image Recognition》获得了CVPR最佳论文。他提出的深度残差网络在2015年可以说是洗刷了图像方面的各大比赛，以绝对优势取得了多个比赛的冠军。而且它在保证网络精度的前提下，将网络的深度达到了152层，后来又进一步加到1000的深度。

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

In [9]:

```
class ResidualBlock(nn.Module):    # 我们定义网络时一般是继承的torch.nn.Module创建新的子类

    def __init__(self, inchannel, outchannel, stride=1):
        super(ResidualBlock, self).__init__()
        #torch.nn.Sequential是一个Sequential容器，模块将按照构造函数中传递的顺序添加到模块中。
        self.left = nn.Sequential(
            nn.Conv2d(inchannel, outchannel, kernel_size=3, stride=stride, padding=1, bias=False),
            # 添加第一个卷积层,调用了nn里面的Conv2d（）
            nn.BatchNorm2d(outchannel), # 进行数据的归一化处理
            nn.ReLU(inplace=True), # 修正线性单元，是一种人工神经网络中常用的激活函数
            nn.Conv2d(outchannel, outchannel, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(outchannel)
        )
        self.shortcut = nn.Sequential()
        if stride != 1 or inchannel != outchannel:
            self.shortcut = nn.Sequential(
                nn.Conv2d(inchannel, outchannel, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(outchannel)
            )
        #  便于之后的联合,要判断Y = self.left(X)的形状是否与X相同

    def forward(self, x): # 将两个模块的特征进行结合，并使用ReLU激活函数得到最终的特征。
        out = self.left(x)
        out += self.shortcut(x)
        out = F.relu(out)
        return out

class ResNet(nn.Module):
    def __init__(self, ResidualBlock, num_classes=10):
        super(ResNet, self).__init__()
        self.inchannel = 64
        self.conv1 = nn.Sequential( # 用3个3x3的卷积核代替7x7的卷积核，减少模型参数
            nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(),
        )
        self.layer1 = self.make_layer(ResidualBlock, 64,  2, stride=1)
        self.layer2 = self.make_layer(ResidualBlock, 128, 2, stride=2)
        self.layer3 = self.make_layer(ResidualBlock, 256, 2, stride=2)
        self.layer4 = self.make_layer(ResidualBlock, 512, 2, stride=2)
        self.fc = nn.Linear(512, num_classes)

    def make_layer(self, block, channels, num_blocks, stride):
        strides = [stride] + [1] * (num_blocks - 1)   #第一个ResidualBlock的步幅由make_layer的函数参数stride指定
        # ，后续的num_blocks-1个ResidualBlock步幅是1
        layers = []
        for stride in strides:
            layers.append(block(self.inchannel, channels, stride))
            self.inchannel = channels
        return nn.Sequential(*layers)

    def forward(self, x):
        out = self.conv1(x)
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = F.avg_pool2d(out, 4)
        out = out.view(out.size(0), -1)
        out = self.fc(out)
        return out


def ResNet18():
    return ResNet(ResidualBlock)
```

训练和测试

In [11]:

```
# 定义是否使用GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# 超参数设置
EPOCH = 5   #遍历数据集次数
pre_epoch = 0  # 定义已经遍历数据集的次数
LR = 0.1        #学习率

# 模型定义-ResNet
net = ResNet18().to(device)

# 定义损失函数和优化方式
criterion = nn.CrossEntropyLoss()  #损失函数为交叉熵，多用于多分类问题
optimizer = optim.SGD(net.parameters(), lr=LR, momentum=0.9, weight_decay=5e-4)
#优化方式为mini-batch momentum-SGD，并采用L2正则化（权重衰减）

# 训练
if __name__ == "__main__":
    print("Start Training, Resnet-18!")
    num_iters = 0
    for epoch in range(pre_epoch, EPOCH):
        print('\nEpoch: %d' % (epoch + 1))
        net.train()
        sum_loss = 0.0
        correct = 0.0
        total = 0
        for i, data in enumerate(trainloader, 0):
            #用于将一个可遍历的数据对象(如列表、元组或字符串)组合为一个索引序列，同时列出数据和数据下标，
            #下标起始位置为0，返回 enumerate(枚举) 对象。

            num_iters += 1
            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()  # 清空梯度

            # forward + backward
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            sum_loss += loss.item() * labels.size(0)
            _, predicted = torch.max(outputs, 1) #选出每一列中最大的值作为预测结果
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
            # 每20个batch打印一次loss和准确率
            if (i + 1) % 20 == 0:
                print('[epoch:%d, iter:%d] Loss: %.03f | Acc: %.3f%% '
                      % (epoch + 1, num_iters, sum_loss / (i + 1), 100. * correct / total))

print("Training Finished, TotalEPOCH=%d" % EPOCH)
```

```
Start Training, Resnet-18!

Epoch: 1
[epoch:1, iter:20] Loss: 823.079 | Acc: 14.336%
[epoch:1, iter:40] Loss: 753.080 | Acc: 16.016%
[epoch:1, iter:60] Loss: 696.070 | Acc: 17.305%
[epoch:1, iter:80] Loss: 651.746 | Acc: 18.887%
[epoch:1, iter:100] Loss: 618.316 | Acc: 20.973%
[epoch:1, iter:120] Loss: 591.565 | Acc: 22.790%
[epoch:1, iter:140] Loss: 570.267 | Acc: 24.470%
[epoch:1, iter:160] Loss: 554.011 | Acc: 25.906%

Epoch: 2
[epoch:2, iter:196] Loss: 415.283 | Acc: 39.648%
[epoch:2, iter:216] Loss: 412.659 | Acc: 39.951%
[epoch:2, iter:236] Loss: 410.153 | Acc: 40.397%
[epoch:2, iter:256] Loss: 407.768 | Acc: 40.894%
[epoch:2, iter:276] Loss: 404.098 | Acc: 41.305%
[epoch:2, iter:296] Loss: 400.821 | Acc: 41.836%
[epoch:2, iter:316] Loss: 397.486 | Acc: 42.215%
[epoch:2, iter:336] Loss: 394.179 | Acc: 42.722%

Epoch: 3
[epoch:3, iter:372] Loss: 345.337 | Acc: 50.234%
[epoch:3, iter:392] Loss: 346.913 | Acc: 50.225%
[epoch:3, iter:412] Loss: 346.748 | Acc: 50.234%
[epoch:3, iter:432] Loss: 342.569 | Acc: 51.021%
[epoch:3, iter:452] Loss: 338.151 | Acc: 51.578%
[epoch:3, iter:472] Loss: 334.200 | Acc: 52.161%
[epoch:3, iter:492] Loss: 331.547 | Acc: 52.539%
[epoch:3, iter:512] Loss: 328.745 | Acc: 53.022%

Epoch: 4
[epoch:4, iter:548] Loss: 296.545 | Acc: 57.617%
[epoch:4, iter:568] Loss: 293.362 | Acc: 58.291%
[epoch:4, iter:588] Loss: 290.632 | Acc: 58.952%
[epoch:4, iter:608] Loss: 286.760 | Acc: 59.590%
[epoch:4, iter:628] Loss: 284.410 | Acc: 59.973%
[epoch:4, iter:648] Loss: 281.798 | Acc: 60.459%
[epoch:4, iter:668] Loss: 281.201 | Acc: 60.572%
[epoch:4, iter:688] Loss: 279.392 | Acc: 60.786%

Epoch: 5
[epoch:5, iter:724] Loss: 253.942 | Acc: 64.316%
[epoch:5, iter:744] Loss: 249.157 | Acc: 64.961%
[epoch:5, iter:764] Loss: 248.299 | Acc: 65.085%
[epoch:5, iter:784] Loss: 247.873 | Acc: 65.229%
[epoch:5, iter:804] Loss: 246.415 | Acc: 65.508%
[epoch:5, iter:844] Loss: 243.419 | Acc: 66.108%
[epoch:5, iter:864] Loss: 241.703 | Acc: 66.353%
Training Finished, TotalEPOCH=5
```