基于 cifar-10 图像分类的

CNN 多模型比较

项目名称：	基于 cifar-10 图像分类的 CNN 多模型比较		
小组成员：	刘泽宇	符莎莎	陈政培
学　　号：	17363058	17363019	17363011
学　　院：	智能工程学院 智能科学与技术专业		
邮　　箱：	1365457082@qq.com		

目录

1. 项目介绍

本小组的期末项目课题为：基于 CIFAR-10 图像分类的 CNN 多模型比较。

在一学期的机器学习课程中，我们学习到了机器学习领域的许多技巧和方法，在这一过程中，我们也对机器学习研究领域的主流问题有了更加清晰的认识。图像分类作为机器学习领域的经典问题与重要研究方向，拥有着非常庞大和充分的方法与研究资料。无论是课堂上学习并实践过的 KNN、SVM、贝叶斯算法等，还是本小组这次主要的研究方向 CNN 多模型网络，都是这一领域发展过程中产生的重要理论与方法。

本小组将课题定为基于 CIFAR-10 图像分类的 CNN 多模型比较，有以下几点原因：

- CIFAR-10 是图像分类领域非常经典和格式清晰的数据集，既有利于我们更深的了解相关概念，又有助于本次项目的实现；
- 神经网络是机器学习领域中，近年来十分火爆的子领域深度学习的核心知识。在课堂上，我们已经对传统机器学习方法有了比较深入和系统的了解，并做了较完备的应用加以理解，而对于深度学习这一火热领域，本小组既好奇，又希望能有深入的了解。而通过复现并优化几个经典网络结构是非常好的理解、学习深度神经网络的方法；
- 通过多模型的比较，我们能够更好地认识到神经网络的演进与优化历史，了解到主流深度学习方法的思考方式与改进方式，对我们这一课堂的知识学习，有着融会贯通、相互促进的作用。

3. 数据准备

3.1 CIFAR10 数据集介绍

- CIFAR-10 是一个更接近普适物体的彩色图像数据集。

  CIFAR-10 是一个用于识别普适物体的小型数据集。共包含 10 个类别的 RGB 彩色图片：飞机（airplane）、汽车（automobile）、鸟类（bird）、猫（cat）、鹿（deer）、狗（dog）、蛙类（frog）、马（horse）、船（ship）和卡车（truck）。图片的尺寸为 32×32，数据集中一共有 50000 张训练图片和 10000 张测试图片。

  CIFAR-10 的图片样例如下所示：

- 对比 MNIST 手写数据集，CIFAR-10 数据集有以下特点：
  - CIFAR-10 是 3 通道的彩色 RGB 图像，而 MNIST 是灰度图像。
  - CIFAR-10 的图片尺寸为 32 × 32，而 MNIST 的图片尺寸为 28 × 28 比 MNIST 稍大。
  - 相比于手写字符， CIFAR-10 含有的是现实世界中真实的物体，不仅

噪声很大，而且物体的比例、特征都不尽相同，这为识别带来很大困难。直接的线性模型如 Softmax 在 CIFAR-10 上表现得很差。

- CIFAR-10 数据集的数据文件名及用途

在 CIFAR-10 数据集中，文件 data_batch_1.bin ~data_batch_5.bin 和 test_batch.bin 中各有 10000 样本。一个样本由 3073 字节组成，第一个字节为标签 label ，剩下 3072 字节为图像数据。样本和样本之间没有多余的字节分割，因此这几个二进制文件的大小都是 30730000 字节。

3.2 数据预处理

项目过程中，我们发现网络训练时会出现过拟合效果。因此，为了提高鲁棒性、优化网络训练效果，我们重新引入了对数据的预处理。具体操作为:

3.2.1 训练集操作优化

transforms 函数作为图像预处理包,其中除了将 PILImage 转变为 FloatTensor 格式以外还有很多与处理功能，所以从中选取了几个比较常见的预处理函数对数据集进行了多样性处理。

1. 裁剪

   加入了随机裁剪 transform.RandomCrop()

2. 翻转和旋转

   加入了依据概率水平翻转 transforms.RandomHorizontalFlip()

3. 图像变换

   加入了标准化 transforms.Normalize() 并且对原有给定的参数 ((0.5,0.5,0.5),(0.5,0.5,0.5)) 结合网络上优秀的学习代码进行了调整

((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))

代码如下图所示：

```
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])
```

3.2.2 优化器改进

原有代码中使用的仍然是 SGD，此处可以修改为优化器中的集大成者 Adam 进一步提升效率和正确率。

介绍：*（参考：https://www.cnblogs.com/yifdu25/p/8183587.html）*

- SGD(Stochastic Gradient Descent，随机梯度下降)：仅仅选取一个样本 j 来求梯度。

$$\theta_i = \theta_i - \alpha(h_\theta(x_0^j, x_1^j, \ldots x_n^j) - y_j)x_i^j$$

- Adam（adaptive moment estimation）是一种可以替代传统随机梯度下降（SGD）过程的一阶优化算法，它能基于训练数据迭代地更新神经网络权重。

  随机梯度下降保持单一的学习率（即 alpha）更新所有的权重，学习率在训练过程中并不会改变。而 Adam 通过计算梯度的一阶矩估计和二阶矩估计而为不同的参数设计独立的自适应性学习率 Adam 算法的提出者描述其为两种随机梯度下降扩展式的优点集合，即：

  - 适应性梯度算法（AdaGrad）为每一个参数保留一个学习率以提升在稀疏梯度（即自然语言和计算机视觉问题）上的性能。

- 均方根传播（RMSProp）基于权重梯度最近量级的均值为每一个参数适应性地保留学习率。这意味着算法在非稳态和在线问题上有很有优秀的性能。

Adam 算法同时获得了 AdaGrad 和 RMSProp 算法的优点。Adam 不仅如 RMSProp 算法那样基于一阶矩均值计算适应性参数学习率，它同时还充分利用了梯度的二阶矩均值（即有偏方差/uncentered variance）。具体来说，算法计算了梯度的指数移动均值（exponential moving average），超参数 beta1 和 beta2 控制了这些移动均值的衰减率。移动均值的初始值和 beta1、beta2 值接近于 1（推荐值），因此矩估计的偏差接近于 0。该偏差通过首先计算带偏差的估计而后计算偏差修正后的估计而得到提升。

相关代码如下图所示：

```
optimizer = optim.Adam(net.parameters(), lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=5e-4)
```

4. 多模型实现与比较

4.1 总体介绍

本小组的项目目标是基于 CIFAR-10 数据集，实现近年来常见的多种网络结构，并在此基础上对这些网络结构加以优化、对比，得到项目成果。

具体来说，我们学习并编写了如下网络结构：

LeNet-5；VGG；ResNet；GoogleNet；DPN 网络。

在实现过程中，由于网络训练时间过长，我们计划实现的另外一些网络结构未能按时得到结果，暂不在此列举。

4.2 网络结构原理

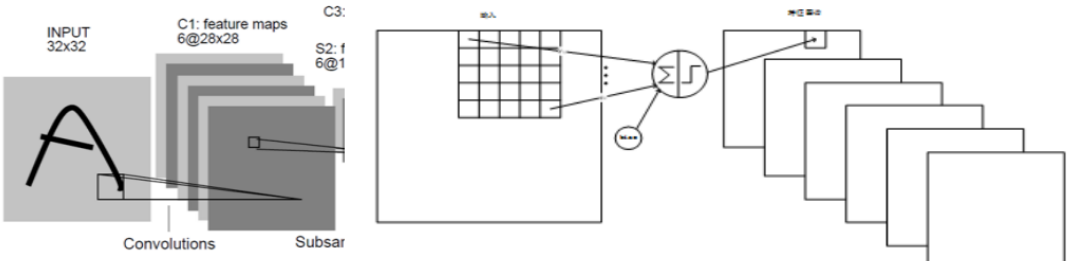
下面对本小组实现网络模型的具体结构与原理进行总结。

4.2.1 LeNet-5

1. 卷积层 C1

C1 层是卷积层，形成 6 个 feature map。卷积核大小 5×5，每个 feature map 内参数共享，即每个 feature map 内只使用一个共同卷积核。卷积核有 5×5 个连接参数加上一个偏置参数共 26 个偏置参数。卷积区域每次滑动 1 个像素，这样卷积层形成的每个 feature map 大小是(32-5)/1+1 = 28×28.

C1 层共有 26×6 = 156 个训练参数，有(5×5+1)×28×28×6 = 122304 个连接。



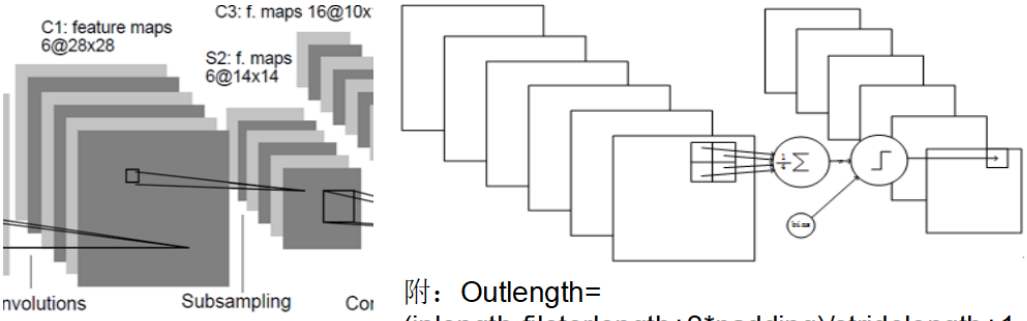
附：Outlength=
(inlength-fileterlength+2*padding)/stridelength+1

2. 池化层 S2

S2 是一个 Pooling 层(下采样)，利用图像局部相关性的原理，对图像进行子抽样,可以减少数据处理量同时保留有用信息。C1 层的 6 个 28×28feature maps 分别进行以 2×2 为单位的下采样得到 6 个 14×14 的图。每个 feature map 使用一个下采样核。

14： (28-2)/2+1

一共 5×14×14×6 = 5880 个连接。(5=2*2+1)



附：Outlength=
(inlength-fileterlength+2*padding)/stridelength+1

3. 卷积层 C3

C3 层是一个卷积层。卷积核同 C1，但每个节点与 S2 中的多个图相连。C3 层有 16 个 10×10（14-5+1）的图。

显然，这证明我们采用了 16 个卷积核，但 C3 层不同的是，其采用了 16 种不同的卷积核，也即参数不共享。C3 层中每个 feature map 由 S2 中所有 6 个或其中几个 feature map 组合而成，不将每个连接到 C3 的每个

feature map 是因为：

- 不完全的连接机制可以将连接数量控制在合理的范围内；
- 破坏了网络的对称性，迫使网络抽取不同的特征。

例如，常用的一个方式是：C3 的前 6 个 feature maps 以 S2 中相邻的 3 个 feature map 作为输入；接下来的 6 个 feature map 以 S2 中相邻的 4 个 feature map 作为输入；然后的 3 个以不相邻的 4 个 feature map 作为输入；最后一个将 S2 中所有 feature map 作为输入，这样 S3 层共有：

（6（3*25+1）+6*（4*25+1）+3*（4*25+1）+（25*6+1）=1516* 个可训练参数和 151600（10*10*1516=151600）个连接。

4. 池化层 S4

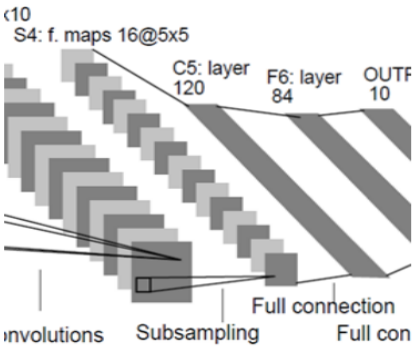
S4 是一个下采样层，C3 的 16 个 10×10 的图分别进行以 2×2 为单位的下采样得到 16 个 5 ((10-2)/2+1) ×5 的图。 S4 层有 32(每个 feature map1 个因子加上一个偏置,16(1+1))个可训练参数和 2000 个连接。(16\(2*2+1)*5*5=2000)

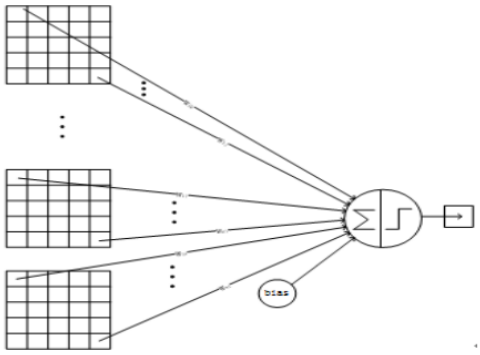
5. 卷积层 C5

存在 120 个 feature maps。每个单元与 S4 层的全部 16 个单元的 5*5 邻域相连。S4 层 feature map 与 filter 大小一样均为 5*5，故 C5 层 feature map 大小为 1*1，构成了 S4 与 C5 之间的全连接。

之所以仍标记为卷积层而非全连接层，是因为该层大小视输入图片大小而定。

每个都与上一层的 16 个图相连。所以共有(5x5x16+1)x120 = 48120 个参数，同样有 48120 个连接。





附：Outlength=
(inlength-fileterlength+2*padding)/stridelength+1

6. 全连接层 F6

全连接层，共有 84 个单元（选择这个数字来自于输出层的设计）与 C5 层全连接。

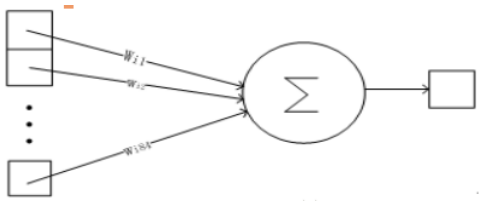
F6 层对应于一个 7×12 的比特图，该层的训练参数和连接数都是 (120×(1×1)+1)×84 = 10164 个。

与经典神经网络相同，F6 层计算输入向量和权重向量之间的点积，再加上一个偏置，然后将其传递给 sigmoid 函数产生单元 i 的一个状态。

7. 全连接层 OutPut

共有 10 个节点，分别代表数字 0-9，如果节点 i 的输出值为 0，则网络识别出的结果是数字 i。采用径向基函数(RBF)的网络连接方式。假设 x 是上一层的输入，y 是 RBF 的输出，则 RBF 的计算方式是：

$$y_i = \sum_j (x_j - w_{ij})^2$$



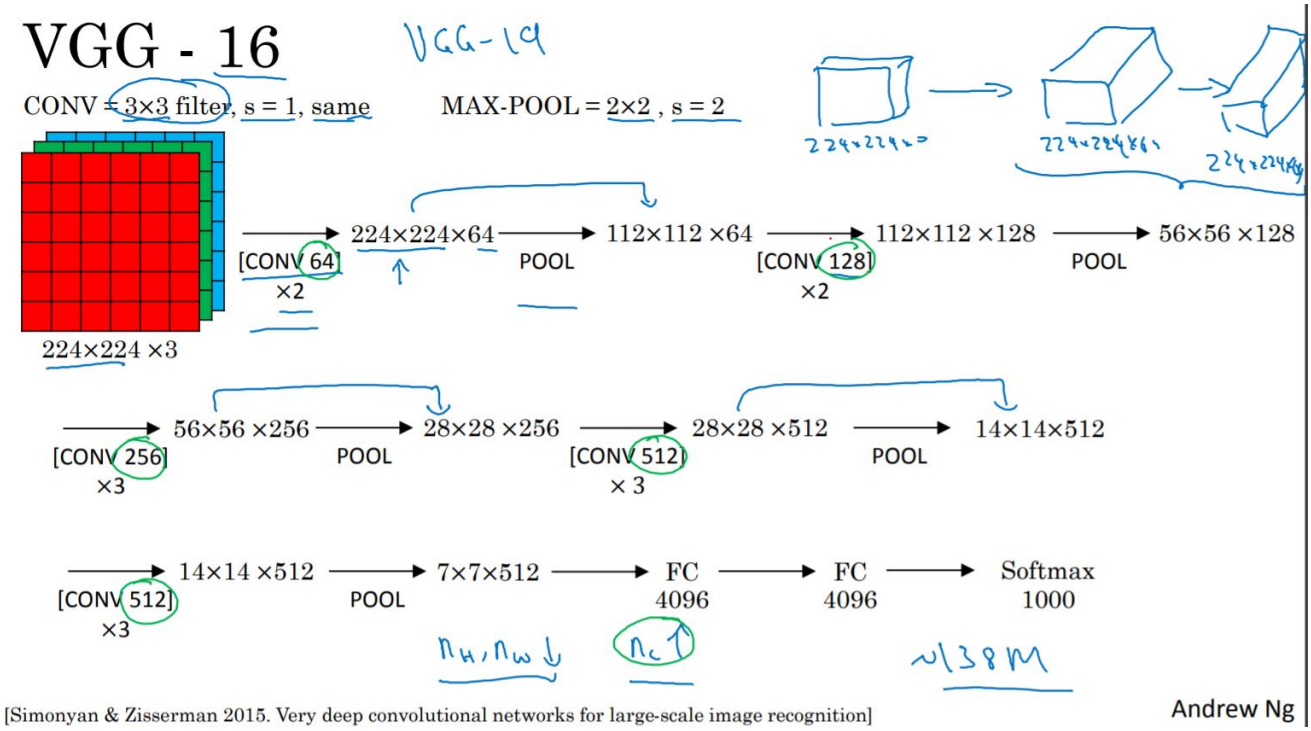
yi 的值由 i 的比特图编码(即参数 Wij)确定，i 从 0-9，j 从 0-7*12-1。

RBF 输出的值越接近于 0，则结果越接近于 i，即越接近于 i 的 ASCII 编

码图，表示当前网络输入的识别结果是字符 i。

该层共有 84×10 = 840 个参数和连接。

4.2.2 VGG



ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Table 2: **Number of parameters** (in millions).

Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

如上图所示，为 VGG 网络结构。

- VGGNet 的特点：
  - VGGNet 经常被用来提取图像特征；而且 VGGNet 训练后的模型参数在其官网开源，可用来在特定的图像分类任务上进行再次训练。
  - VGG 的网络结构比较简单，由 5 段卷积，3 个全连接层和 softmax 输出层组成，结构非常简洁。层与层之间使用 maxPooling 分开，同样采 ReLu 作为激活函数。VGGNet 全部使用 3x3 的卷积核与 2x2 的池化核，主要通过不断加深网络结构来提升性能。
  - 小卷积核，多卷积层；
  - VGGNet 有 5 段卷积，每段有 2，3，4 个卷积层，每段内部的卷积核数量都相同，且越到后面层数越多：64->128->256-512。

- ✧ 每个卷积层都是使用的 3x3 的卷积核，且采用多个 3x3 卷积层堆在一起的设计，这种设计是非常有用的。都知道，两个 3x3 的卷积层串联起来卷积的结果相当于一个 5x5 卷积层卷积的结果，三个 3x3 卷积层卷积的结果相当于一个 7x7 的卷积层卷积的结果。如果采用 3 个 3x3 的卷积层，只需要 3x3x3 = 27 个参数，而 1 个 7x7 需要 49 个参数；与此同时，采用 3 个 3x3 的卷积层会进行 3 次非线性变换(经过 3 次 ReLu 激活函数)，而后者只进行一次非线性变换，因此采用多层 3x3 卷积层能让网络对特征的学习能力更强。

➢ 网络结构具体描述如下：

- ✧ 输入 224x224x3 的图片，经过 64 个卷积核的两次卷积，卷积 filter 为 3x3，stride=1,padding=1； 此时 out_size=(224-3+1*2)/1 + 1 = 224;两轮卷积后 shape=（224,244,64）
- ✧ 采用一次 Pooling, stride=2, padding=0, filter=2， 此时 out_size=(224-2)/2 + 1 = 112;shape=(112,112,64)
- ✧ 两层 channel=128 的卷积： out_size=(112-3+2)/1 + 1 = 112;两轮卷积后 shape=(112,112,128)
- ✧ pooling：56x56x128
- ✧ conv256x3： 56x56x256
- ✧ pooling： 28x28x256
- ✧ conv512x3： 28x28x512
- ✧ pooling： 14x14x512
- ✧ conv512x3： 14x14x512

- ✧ pooling: 7x7x512
- ✧ FC: 4096, 此时，共有 7x7x512=25088
- ✧ FC：4096
- ✧ Softmax：1000（替换为 10 即可）

➢ VGGNet 的缺点：

- ✧ 层数太多，卷积核小的话会增加计算量，运算时需要大量的内存(或显存)，耗费资源，训练时间长等。

4.2.3 GoogLeNet

（参考：https://my.oschina.net/u/876354/blog/1637819）

VGG 继承了 LeNet 以及 AlexNet 的一些框架结构，而 GoogLeNet 则做了更加大胆的网络结构尝试。深度只有 22 层，而大小比 AlexNet 和 VGG 小很多：GoogleNet 参数为 500 万个，AlexNet 参数个数是 GoogLeNet 的 12 倍，VGGNet 参数又是 AlexNet 的 3 倍，因此在内存或计算资源有限时，GoogleNet 是比较好的选择；另外，从模型结果来看，GoogLeNet 的性能却更加优越。

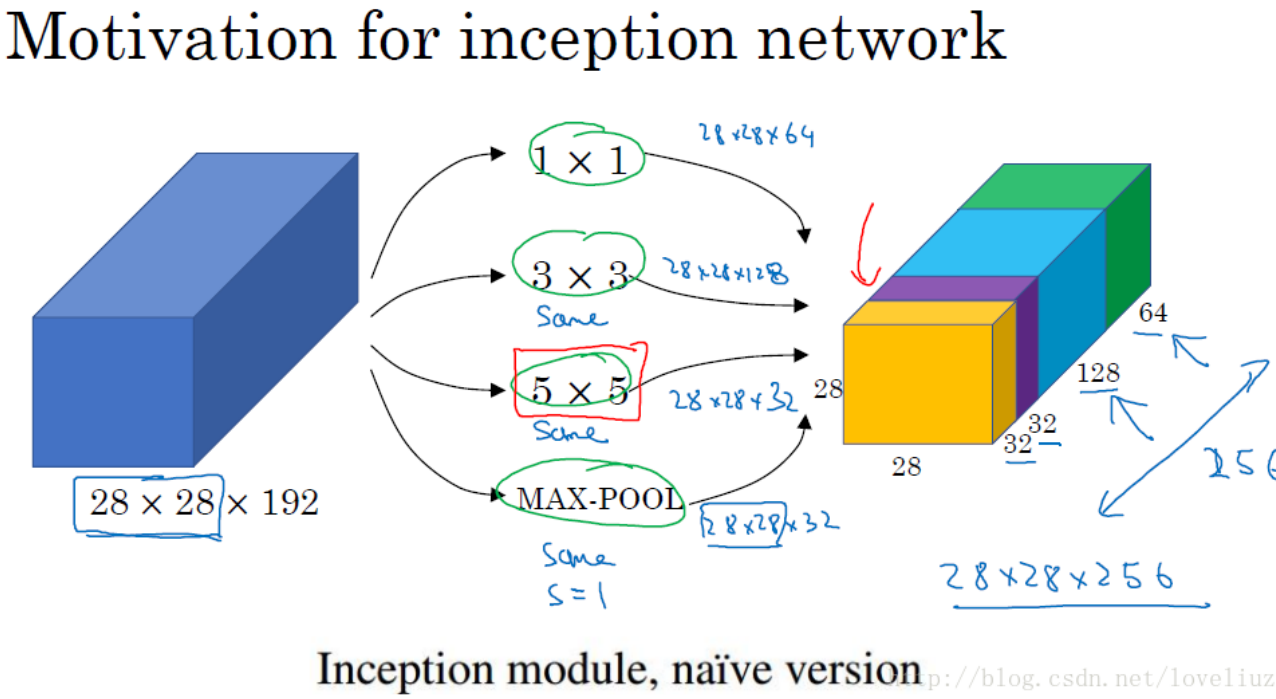
1. GoogLeNet 提升性能的方法：

一般来说，提升网络性能最直接的办法就是增加网络深度和宽度，深度指网络层次数量、宽度指神经元数量。但这种方式存在以下问题：

- 参数太多，如果训练数据集有限，很容易产生过拟合；
- 网络越大、参数越多，计算复杂度越大，难以应用；
- 网络越深，容易出现梯度弥散问题（梯度越往后穿越容易消失），难以优化模型。

GoogLeNet 团队提出了 Inception 网络结构，就是构造一种“基础神经元”结构，来搭建一个稀疏性、高计算性能的网络结构，其方法既能保持网络结构的稀疏性，又能利用密集矩阵的高计算性能。

2. Inception 模块介绍



Inception module, naïve version

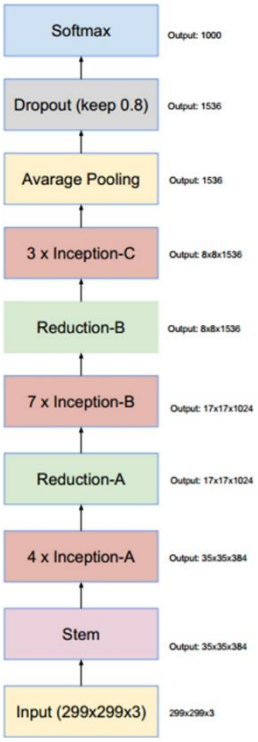
- 该结构将 CNN 中常用的卷积（1x1，3x3，5x5）、池化操作（3x3）堆叠在一起（卷积、池化后的尺寸相同，将通道相加），一方面增加了网络的宽度，另一方面也增加了网络对尺度的适应性。

  网络卷积层中的网络能够提取输入的每一个细节信息，同时 5x5 的滤波器也能够覆盖大部分接受层的输入。还可以进行一个池化操作，以减少空间大小，降低过度拟合。在这些层之上，在每一个卷积层后都要做一个 ReLU 操作，以增加网络的非线性特征。

- Inception 的作用：代替人工确定卷积层中的过滤器类型或者确定是否需要创建卷积层和池化层，即：不需要人为的决定使用哪个过滤器，

是否需要池化层等，由网络自行决定这些参数，可以给网络添加所有可能值，将输出连接起来，网络自己学习它需要什么样的参数。

- 需要注意的是，在 Inception V4 结构中，结合了残差神经网络 ResNet。



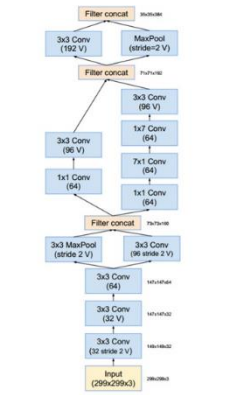


Figure 3. The schema for stem of the pure Inception-v4 and Inception-ResNet-v2 networks. This is the input part of those networks. Cf. Figures 9 and 15

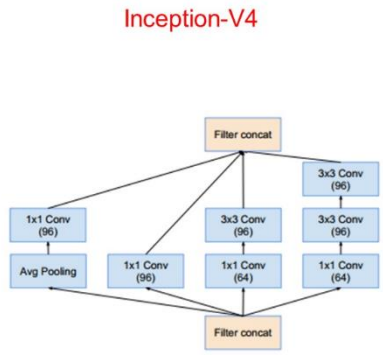


Figure 4. The schema for 35 × 35 grid modules of the pure Inception-v4 network. This is the Inception-A block of Figure 9.

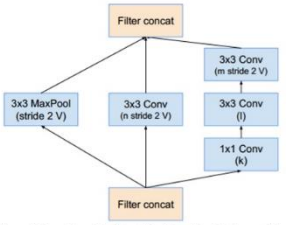


Figure 7. The schema for 35 × 35 to 17 × 17 reduction module. Different variants of this blocks (with various number of filters) are used in Figure 9, and 15 in each of the new Inception(-v4, -ResNet-v1, -ResNet-v2) variants presented in this paper. The k, l, m, n numbers represent filter bank sizes which can be looked up in Table 1.

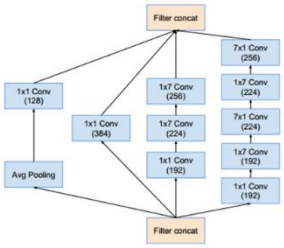


Figure 5. The schema for 17 × 17 grid modules of the pure Inception-v4 network. This is the Inception-B block of Figure 9.

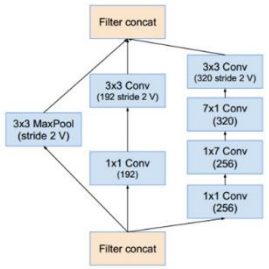


Figure 8. The schema for 17 × 17 to 8 × 8 grid-reduction module. This is the reduction module used by the pure Inception-v4 network in Figure 9.

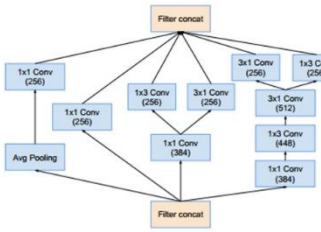


Figure 6. The schema for 8 × 8 grid modules of the pure Incept v4 network. This is the Inception-C block of Figure 9.

关于 ResNet，将在下面做专门的介绍。

由于该部分的实现主要基于 GoogLeNet 相关代码及上方链接，故不在此进一步赘述。

4.2.4 ResNet

在`VGG`网络不断在层数中越走越深，发现深度 CNN 网络达到一定的深度之后再一味地增加层数并不会进一步提高分类性能，反而会导致网络收敛变得很慢。而`Resnet`代替`VGG`成为了一般计算机视觉图像领域问题中的基础特征提取网络。

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

- 残差

  ResNets v1 堆叠了很多残差单元，每一个单元能够写为下面的通用形式：

$$y_l = h(x_l) + \mathcal{F}(x_l, \mathcal{W}_l)$$

$$x_{l+1} = f(y_l)$$

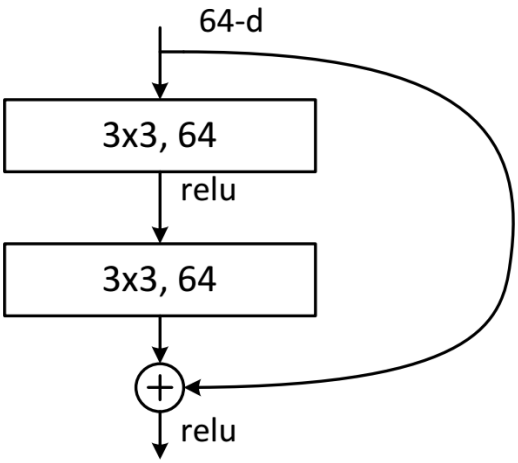
  这里 x_l 和 x_{l+1} 是第 l 个单元的输入和输出，F 是一个残差函数。在 ResNet 中，h(xl)=xl 是一个 identity mapping，f 是一个 ReLU 函数。

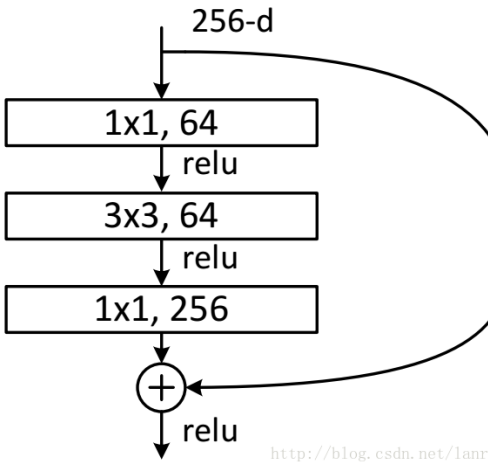
- 如何解决“随着网络加深，准确率不下降”的问题?

  理论上，对于“随着网络加深，准确率下降”的问题，Resnet 提供了两种选择方式，也就是 identity mapping 和 residual mapping。如果网络已经到达最优，继续加深网络，residual mapping 将被 push 为 0，只剩下 identity mapping，这样理论上网络一直处于最优状态了，网络的性能也就不会随着深度增加而降低了。

  实验上很好地佐证了理论结果。

- ResNet 结构





论文中提出了两种 ResNet 设计方式。

这两种结构分别针对 ResNet34（左图）和 ResNet50/101/152（右图），一般称整个结构为一个”building block“。其中右图又称为”bottleneck design”,目的一目了然，就是为了降低参数的数目，第一个 1x1 的卷积把 256 维 channel 降到 64 维，然后在最后通过 1x1 卷积恢复，整体上用的参数数目：1x1x256x64 + 3x3x64x64 + 1x1x64x256 = 69632，而不使用 bottleneck 的话就是两个 3x3x256 的卷积，参数数目: 3 x 3 x 256 x 256 x 2 = 1179648，差了 16.94 倍。

对于常规 ResNet，可以用于 34 层或者更少的网络中，对于 Bottleneck Design 的 ResNet 通常用于更深的如 101 这样的网络中，目的是减少计算和参数量（实用目的）。

4.2.5 DPN

`DPN`网络是将`ResNeXt`和`DenseNet`网络融合，理论上对特征的利用更加充分。

DPN 和 ResNeXt（ResNet）的结构很相似。最开始是 7*7 的卷积层和 max pooling 层，然后是 4 个 stage，每个 stage 包含几个 sub-stage，再接着是一

个 global average pooling 和全连接层，最后是 softmax 层。重点在于 stage 里面的内容，也是 DPN 算法的核心。

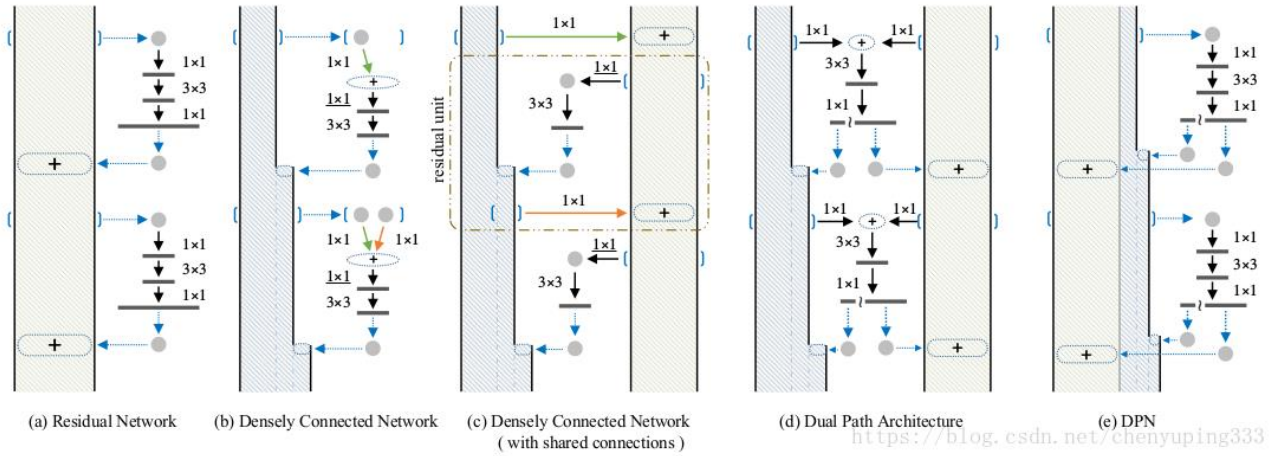
Table 1: Architecture and complexity comparison of our proposed Dual Path Networks (DPNs) and other state-of-the-art networks. We compare DPNs with two baseline methods: DenseNet [5] and ResNeXt [21]. The symbol $(+k)$ denotes the width increment on the densely connected path.

stage	output	DenseNet-161 (k=48)	ResNeXt-101 (32×4d)	ResNeXt-101 (64×4d)	DPN-92 (32×3d)	DPN-98 (40×4d)
conv1	112x112	7 × 7, 96, stride 2	7 × 7, 64, stride 2	7 × 7, 64, stride 2	7 × 7, 64, stride 2	7 × 7, 96, stride 2
conv2	56x56	3 × 3 max pool, stride 2	3 × 3 max pool, stride 2	3 × 3 max pool, stride 2	3 × 3 max pool, stride 2	3 × 3 max pool, stride 2
		$\begin{bmatrix} 1\times1, 192 \\ 3\times3, 48 \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128, G=32 \\ 1\times1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256, G=64 \\ 1\times1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 96 \\ 3\times3, 96, G=32 \\ 1\times1, 256\ (+16) \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 160 \\ 3\times3, 160, G=40 \\ 1\times1, 256\ (+16) \end{bmatrix} \times 3$
conv3	28×28	$\begin{bmatrix} 1\times1, 192 \\ 3\times3, 48 \end{bmatrix} \times 12$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256, G=32 \\ 1\times1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512, G=64 \\ 1\times1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times1, 192 \\ 3\times3, 192, G=32 \\ 1\times1, 512\ (+32) \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times1, 320 \\ 3\times3, 320, G=40 \\ 1\times1, 512\ (+32) \end{bmatrix} \times 6$
conv4	14×14	$\begin{bmatrix} 1\times1, 192 \\ 3\times3, 48 \end{bmatrix} \times 36$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512, G=32 \\ 1\times1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1\times1, 1024 \\ 3\times3, 1024, G=64 \\ 1\times1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1\times1, 384 \\ 3\times3, 384, G=32 \\ 1\times1, 1024\ (+24) \end{bmatrix} \times 20$	$\begin{bmatrix} 1\times1, 640 \\ 3\times3, 640, G=40 \\ 1\times1, 1024\ (+32) \end{bmatrix} \times 20$
conv5	7×7	$\begin{bmatrix} 1\times1, 192 \\ 3\times3, 48 \end{bmatrix} \times 24$	$\begin{bmatrix} 1\times1, 1024 \\ 3\times3, 1024, G=32 \\ 1\times1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 2048 \\ 3\times3, 2048, G=64 \\ 1\times1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 768 \\ 3\times3, 768, G=32 \\ 1\times1, 2048\ (+128) \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 1280 \\ 3\times3, 1280, G=40 \\ 1\times1, 2048\ (+128) \end{bmatrix} \times 3$
	1×1	global average pool 1000-d fc, softmax	global average pool 1000-d fc, softmax	global average pool 1000-d fc, softmax	global average pool 1000-d fc, softmax	global average pool 1000-d fc, softmax
# params		$\mathbf{28.9} \times 10^6$	$\mathbf{44.3} \times 10^6$	$\mathbf{83.7} \times 10^6$	$\mathbf{37.8} \times 10^6$	$\mathbf{61.7} \times 10^6$
FLOPs		$\mathbf{7.7} \times 10^9$	$\mathbf{8.0} \times 10^9$	$\mathbf{15.5} \times 10^9$	$\mathbf{6.5} \times 10^9$	$\mathbf{11.7} \times 10^9$

DPN 算法简单讲就是将 ResNeXt 和 DenseNet 融合成一个网络，首先介绍二者相关内容。

- 是 ResNet 的某个 stage 中的一部分。(a) 左边的大矩形框表示输入输出内容，对一个输入 x，分两条线走：一条线还是 x 本身，另一条线是 x 经过 1*1 卷积，3*3 卷积，1*1 卷积（这三个卷积层的组合又称作 bottleneck），然后把这两条线的输出做一个 element-wise addition，也就是对应值相加，就是（a）中的加号，得到的结果又变成下一个同样模块的输入，几个这样的模块组合在一起就成了一个 stage（比如 Table1 中的 conv3）。
- （b）表示 DenseNet 的核心内容。(c) 的左边竖着的多边形框表示输入输出内容，对输入 x，只走一条线，那就是经过几层卷积后和 x 做一个通道的合并（cancat），得到的结果又成了下一个小模块的输入，这样每一个小模块的输入都

在不断累加，举个例子：第二个小模块的输入包含第一个小模块的输出和第一个小模块的输入，以此类推。



DPN 则是将 Residual Network 和 Densely Connected Network 融合在一起。（e）中竖着的矩形框和多边形框的含义和前面一样。具体在代码中，对于一个输入 x 分两种情况：

- 如果 x 是整个网络第一个卷积层的输出或者某个 stage 的输出，会对 x 做一个卷积，然后做 slice，也就是将输出按照 channel 分成两部分：data_o1 和 data_o2，可以理解为（e）中竖着的矩形框和多边形框；
- 另一种是在 stage 内部的某个 sub-stage 的输出，输出本身就包含两部分：data_o1 和 data_o2)。走两条线：一条线是保持 data_o1 和 data_o2 本身，和 ResNet 类似；另一条线是对 x 做 1*1 卷积，3*3 卷积，1*1 卷积，然后再做 slice 得到两部分 c1 和 c2，最后 c1 和 data_o1 做相加（element-wise addition）得到 sum，类似 ResNet 中的操作；c2 和 data_o2 做通道合并（concat）得到 dense（这样下一层就可以得到这一层的输出和这一层的输入），也就是最后返回两个值：sum 和 dense。

以上这个过程就是 DPN 中 一个 stage 中的一个 sub-stage。有两个细节，一个是 3*3 的卷积采用的是 group 操作，类似 ResNeXt，另一个是在每个 sub-stage 的首尾都会对 dense 部分做一个通道的加宽操作。

5. 建模及部分代码展示

1. 载入数据集

```
In [2]: data_transform = transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize((0.4914, 0.4822, 0.4465), (0.229, 0.224, 0.225))
        ])

        trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True,
                                                transform=data_transform)
        testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True,
                                               transform=data_transform)

        trainloader = torch.utils.data.DataLoader(
            trainset,        #数据集
            batch_size=16,   #批处理大小，若样本数不能被batch_size整除，最后剩余多少使用多少
            shuffle=True,    #随机打乱顺序
            num_workers=2)   #多线程读取数据的线程数
        testloader = torch.utils.data.DataLoader(testset, batch_size=4, shuffle=True, num_wo

        classes = ('plane', 'car', 'bird', 'cat',
                   'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

由于数据集均为 torch.utils.data.Dataset 的子类，因此可以调用 torch.utils.data.DataLoader 使用多线程。

数据集的共同参数：

- transform: 一个函数，原始图片作为输入，返回转换后的图片；即，对 PIL.Image 的转换。根据我们的定义决定 transform 参数的内容
- target transform：输入为 target，输出位对其的转换。例如，输入为图片标注的 string，输出为索引。
- train=True,表示为训练集，否则为测试集；

结果：

```
Files already downloaded and verified
Files already downloaded and verified
```

2. 定义 VGG-16 网络

In [4]:
```
class VGG_16(nn.Module):
    def __init__(self):
        super(VGG_16,self).__init__()#继承父类
        self.Conv1_64 = nn.Conv2d(in_channels=3, out_channels=64, stride=1,
                                  kernel_size=3, padding=1)#(32-3+2)/1 + 1 = 32
        self.Conv2_64 = nn.Conv2d(in_channels=64, out_channels=64, stride=1,
                                  kernel_size=3, padding=1)
        self.Pool = nn.MaxPool2d(kernel_size=2, stride=2)#(32-2+0)/2 + 1 = 16x16x64
        self.Conv1_128 = nn.Conv2d(in_channels=64, out_channels=128, stride=1,
                                   kernel_size=3, padding=1)
        self.Conv2_128 = nn.Conv2d(in_channels=128, out_channels=128, stride=1,
                                   kernel_size=3, padding=1)#16x16x128 ——> 8x8x128
        self.Conv1_256 = nn.Conv2d(in_channels=128, out_channels=256, stride=1,
                                  kernel_size=3, padding=1)
        self.Conv2_256 = nn.Conv2d(in_channels=256, out_channels=256, stride=1,
                                   kernel_size=3, padding=1)#8x8x256 ——> 4x4x256
        self.Conv1_512 = nn.Conv2d(in_channels=256, out_channels=512, stride=1,
                                   kernel_size=3, padding=1)#4x4x512 此时停止pooling
        self.Conv2_512 = nn.Conv2d(in_channels=512, out_channels=512, stride=1,
                                   kernel_size=3, padding=1)
        self.Conv3_512 = nn.Conv2d(in_channels=512, out_channels=512, stride=1,
                                   kernel_size=3, padding=1)
        self.FC1 = nn.Linear(4 * 4 * 512, 1024)
        self.FC2 = nn.Linear(1024, 1024)
        self.FC3 = nn.Linear(1024, 10)#输出output
```

其前向传播过程如图所示：

```
    def forward(self, x):
        x = self.Pool(F.relu(self.Conv2_64(self.Conv1_64(x))))#relu参数inplace=False，表示不改变输入数据（即input中的负值部分不改变）
        x = self.Pool(F.relu(self.Conv2_128(self.Conv1_128(x))))
        x = self.Pool(F.relu(self.Conv2_256(self.Conv1_256(x))))
        x = F.relu(self.Conv3_512(self.Conv2_512(self.Conv1_512(x))))#卷积操作三次
        x = x.view(-1, 4*4*512)#拉伸
        x = F.relu(self.FC1(x))
        #x = F.dropout(x, p=0.3)
        x = F.relu(self.FC2(x))
        x = self.FC3(x)
        return x
net = VGG_16().to(DEVICE)
#net = VGG_16()
#后面加入Dropout层，先试验效果
```

3. 定义 ResNet 网络

In [5]:
```
class ResNetesidualBlock(nn.module):
    def __init__(self, inchannel, outchannel, stride=1):
        super(ResidualBlock, self).__init__()
        '''
        顺序容器，结构化、快速搭建网络。
        '''
        self.left = nn.Sequential(
            nn.Conv2d(inchannel, outchannel, kernel_size=3, stride=stride, padding=1, bias=False),
            nn.BatchNorm2d(outchannel),#归一化处理
            nn.ReLU(inplace=True),
            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)
            )

    def forward(self, x):
        out = self.left(x)
        out += self.shortcut(x) #H(s) = F(x) + 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(
            nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(),
        )
```

4. 网络训练

3. 训练网络

In [ ]:
```
for epoch in range(2):
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        '''
        enumerate() 函数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，一般用在 for 循环当中。
        '''
        inputs, labels = data#获取输入
        inputs = inputs.to(DEVICE)
        labels = labels.to(DEVICE)
        #print(inputs, labels)
        optimizer.zero_grad()#梯度置为0，也就是把loss关于weight的导数变成0.
        #当网络参量进行反馈时，梯度是被积累的而不是被替换掉；但是在每一个batch时毫无疑问并不需要将两个batch的梯度混合起来累积，
        #因此这里就需要每个batch设置一遍zero_grad 了。
        outputs = net(inputs)#将每一batch的数据input进网络
        loss = criterion(outputs, labels).to(DEVICE)#损失函数输入outputs以及真实labels，从而得到损失函数值
        #loss = criterion(outputs, labels)
        loss.backward()# Back Propagation
        optimizer.step()# Gardient Descent,即对参数进行更新

        #打印相关信息
        running_loss += loss.item()
        if i % 100 == 99:
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i+1, running_loss/100))
            running_loss = 0.0
#outputs = nn.Softmax(dim=1)(outputs)
print('训练结束！')
```

5. 在测试集上测试网络

4. 在测试集上测试网络

在整个训练集上进行了2次训练，但是我们需要检查网络是否从数据集中学习到有用的东西。 通过预测神经网络输出的类别标签与实际情况标签进行对比来进行检测。 如果预测正确，我们把该样本添加到正确预测列表。 第一步，显示测试集中的图片并熟悉图片内容。

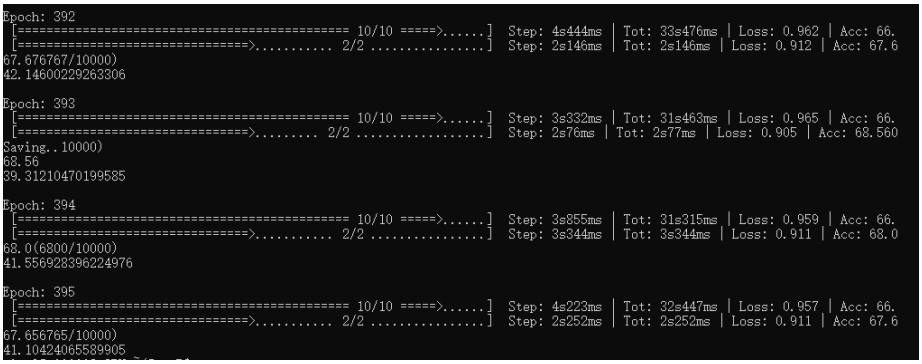
In [ ]:
```
dataiter = iter(testloader)
images, labels = dataiter.next()

imshow(torchvision.utils.make_grid(images))
print('真实标签: ', ' '.join('%5s' % classes[labels[j]] for j in range(4)))
```

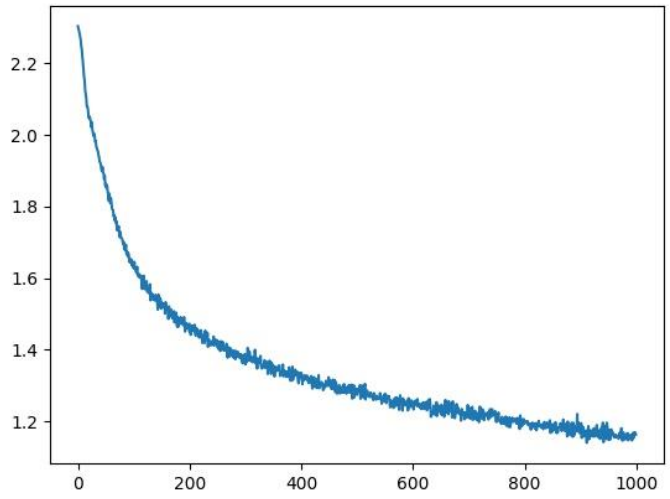
6. 模型评估

6.1 LeNet-5

- 直接将 batch_size 设置为 5000，以比较快的速度进行训练，平均每轮训练耗时`41s`；
- 在进行了 400 次训练后,训练集正确率只有 66%,测试集正确率仅有 67%;
- 且进一步的训练也并没有办法让`Lenet`的正确率突破 70%。



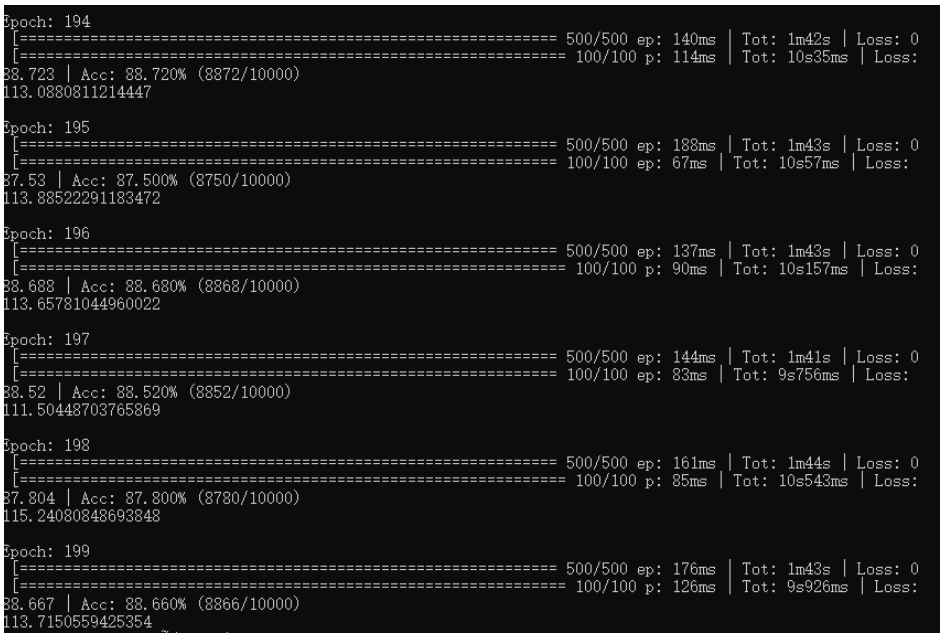
训练结果



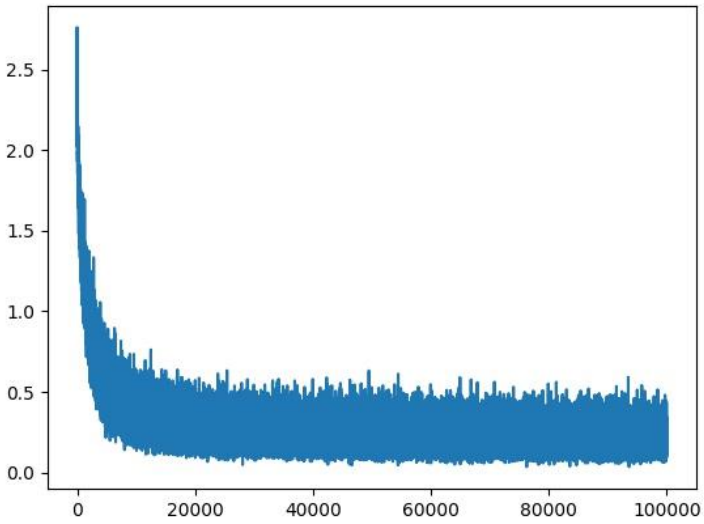
Loss 曲线

6.2 VGG

- 占用内存`4G`左右，batch_size 设置为 100，平均每轮训练用时`115s`；
- 在完成 200 次训练后，训练集正确率达到 88%，测试集正确率在 88%



训练结果



Loss 曲线

6.3 ResNet

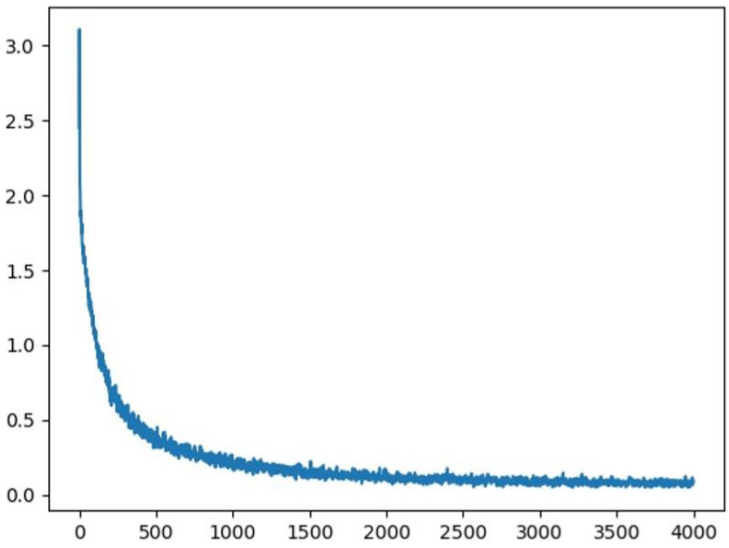
- 占用内存`7G`左右，平均每轮训练用时`690s`；

- 在完成 200+次训练后，训练集正确率达到 98.036%，测试集正确率在 90.27%

```
Epoch: 184
 [========================== 40/40 ==========================>.] Step: 483ms | Tot: 18s913ms | Loss: 0.059 | Acc: 98.006% (49003/50000)
 [========================== 20/20 ==========================>...] Step: 48ms | Tot: 1s475ms | Loss: 0.355 | Acc: 90.860% (9086/10000)
90.86
22.451377868652344

Epoch: 185
 [========================== 40/40 ==========================>.] Step: 483ms | Tot: 18s996ms | Loss: 0.059 | Acc: 98.036% (49018/50000)
 [========================== 20/20 ==========================>...] Step: 49ms | Tot: 1s236ms | Loss: 0.399 | Acc: 90.270% (9027/10000)
90.27
```

训练结果



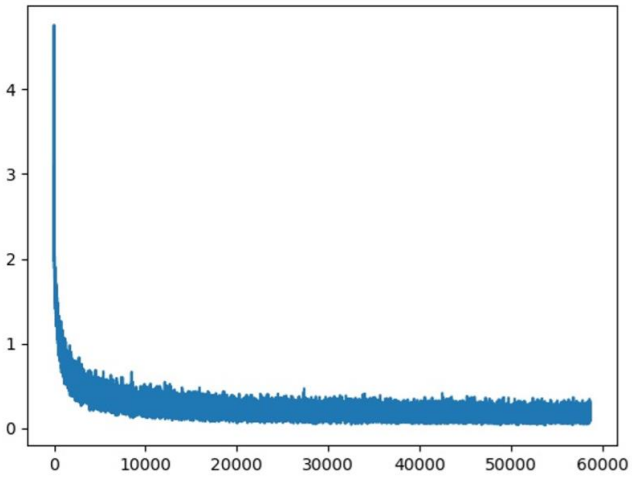
Loss 曲线

6.4 DPN

- `batch_size`设置 128，占用内存`10G`左右，平均每轮训练用时`228s`；
- 在完成 150 次训练后（其中包含未保存的负训练参数 10 次左右），训练集正确率达到 94.59%，测试集正确率在 91.04%。

```
Epoch: 159
 [========================== 391/391 ==========================>] Step: 371ms | Tot: 3m33s | Loss: 0.154 | Acc: 94.662% (47331/50000)
 [========================== 100/100 ==========================>] Step: 79ms | Tot: 8s | Loss: 0.334 | Acc: 89.610% (8961/10000)
89.61
228.54441571235657
(base) sysuai@room:/public/data/sysuai/CrysR$ python main.py
==> Preparing data..
Files already downloaded and verified
Files already downloaded and verified
==> Building model..
==> Resuming from checkpoint..
 [========================== 100/100 ==========================>] Step: 176ms | Tot: 10s846ms | Loss: 0.303 | Acc: 91.040% (9104/10000)
91.04
14.855919122695923
```

训练结果



Loss 曲线

6.5 GoogLeNet

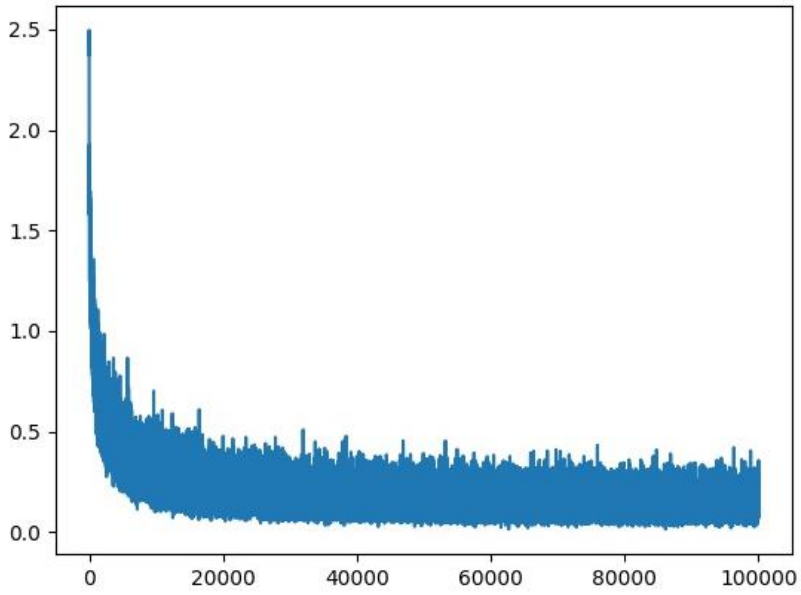
```
Epoch: 364
 [========================================== 500/500 ==============>]  Step: 646ms | Tot: 4m32s | Loss: 0.129 | Acc: 95.5
 [========================================== 100/100 ============>]  Step: 149ms | Tot: 17s559ms | Loss: 0.390 | Acc: 8
88.84% (8884/10000)
291.00702810287476

Epoch: 365
 [========================================== 500/500 ==============>]  Step: 616ms | Tot: 4m34s | Loss: 0.124 | Acc: 95.7
 [========================================== 100/100 ============>]  Step: 157ms | Tot: 16s865ms | Loss: 0.335 | Acc: 8
89.40% (8940/10000)
291.6140196323395

Epoch: 366
 [========================================== 500/500 ==============>]  Step: 464ms | Tot: 4m34s | Loss: 0.128 | Acc: 95.6
 [========================================== 100/100 ============>]  Step: 150ms | Tot: 18s466ms | Loss: 0.381 | Acc: 8
88.79% (8879/10000)
293.78067898750305

Epoch: 367
 [========================================== 500/500 ==============>]  Step: 465ms | Tot: 4m34s | Loss: 0.129 | Acc: 95.5
 [========================================== 100/100 ============>]  Step: 205ms | Tot: 18s14ms | Loss: 0.441 | Acc: 87
87.69 (8769/10000)
293.3742346763611
```

训练结果



Loss 曲线

- `batch_size`设置 100，占用内存`7G`左右，平均每轮训练用时`290s`在完成 400 次训练后（其中最后一次更优的参数在第 300 次训练）；
- 训练集正确率达到 95.54%，测试集正确率在 90.17%。

7. 模型应用

1. Lenet5`在`cifar-10`上的表现就体现出这个简单的网络结构的局限性了，而且loss 的收敛速度相比其他更先进的结构也显著缓慢；
2. 在经过诸如训练集的重组、优化器的改进之后，有效的让有限的训练集实现更大的训练价值，而且使用了`Adam`之后，`Resnet`和`DPN`的正确率都有轻微的提高；
3. 可见`Resnet`已经是一个效果不错的 CNN 结构了；而`DPN`作为在此基础上的进一步优秀算法融合，明显网络结构更加复杂了，仅仅 128 的`batch_size`就占用了`10GB`的显存；参数的复杂多样，反馈机制的丰富，让`DPN`能够在更少的重复训练次数中取得更加优异的表现。

8. 论文参考

1. DPN：https://arxiv.org/abs/1707.01629
2. DPN code：https://github.com/rwightman/pytorch-dpn-pretrained
3. GoogLeNet v4：http://arxiv.org/abs/1602.07261
4. GoogLeNet v3：http://arxiv.org/abs/1512.00567
5. GoogLeNet v2：http://arxiv.org/abs/1502.03167
6. GoogLeNet v1：http://arxiv.org/abs/1409.4842
7. GoogLeNet code：https://github.com/jiawen9611/classification_Pytorch_Proj
8. ResNet：https://arxiv.org/abs/1512.03385
9. ResNet code：https://github.com/KaimingHe/deep-residual-networks