

`tf.nn.max_pool` 是 TensorFlow 中的最大池化函数，我们这里使用 2×2 的最大池化，即将一个 2×2 的像素块降为 1×1 的像素。最大池化会保留原始像素块中灰度值最高的那个像素，即保留最显著的特征。因为希望整体上缩小图片尺寸，因此池化层的 `strides` 也设为横竖两个方向以 2 为步长。如果步长还是 1，那么我们会得到一个尺寸不变的图片。

```
def conv2d(x, W):
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1],
                         padding='SAME')
```

在正式设计卷积神经网络的结构之前，先定义输入的 `placeholder`，`x` 是特征，`y_` 是真实的 `label`。因为卷积神经网络会利用到空间结构信息，因此需要将 1D 的输入向量转为 2D 的图片结构，即从 1×784 的形式转为原始的 28×28 的结构。同时因为只有一个颜色通道，故最终尺寸为 $[-1, 28, 28, 1]$ ，前面的 -1 代表样本数量不固定，最后的 1 代表颜色通道数量。这里我们使用的 tensor 变形函数是 `tf.reshape`。

```
x = tf.placeholder(tf.float32, [None, 784])
y_ = tf.placeholder(tf.float32, [None, 10])
x_image = tf.reshape(x, [-1, 28, 28, 1])
```

接下来定义我们的第一个卷积层。我们先使用前面写好的函数进行参数初始化，包括 `weights` 和 `bias`，这里的 $[5, 5, 1, 32]$ 代表卷积核尺寸为 5×5 ，1 个颜色通道，32 个不同的卷积核。然后使用 `conv2d` 函数进行卷积操作，并加上偏置，接着再使用 `ReLU` 激活函数进行非线性处理。最后，使用最大池化函数 `max_pool_2x2` 对卷积的输出结果进行池化操作。

```
W_conv1 = weight_variable([5, 5, 1, 32])
b_conv1 = bias_variable([32])
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1)
```

现在定义第二个卷积层，这个卷积层基本和第一个卷积层一样，唯一的不同是，卷积核的数量变成了 64，也就是说这一层的卷积会提取 64 种特征。

```
W_conv2 = weight_variable([5, 5, 32, 64])
```

```
b_conv2 = bias_variable([64])
h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_pool2 = max_pool_2x2(h_conv2)
```

因为前面经历了两次步长为 2×2 的最大池化，所以边长已经只有 $1/4$ 了，图片尺寸由 28×28 变成了 7×7 。而第二个卷积层的卷积核数量为 64，其输出的 tensor 尺寸即为 $7 \times 7 \times 64$ 。我们使用 `tf.reshape` 函数对第二个卷积层的输出 tensor 进行变形，将其转成 1D 的向量，然后连接一个全连接层，隐含节点为 1024，并使用 ReLU 激活函数。

```
W_fc1 = weight_variable([7 * 7 * 64, 1024])
b_fc1 = bias_variable([1024])
h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)
```

为了减轻过拟合，下面使用一个 Dropout 层，Dropout 的用法第 4 章已经讲过，是通过一个 placeholder 传入 `keep_prob` 比率来控制的。在训练时，我们随机丢弃一部分节点的数据来减轻过拟合，预测时则保留全部数据来追求最好的预测性能。

```
keep_prob = tf.placeholder(tf.float32)
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
```

最后我们将 Dropout 层的输出连接一个 Softmax 层，得到最后的概率输出。

```
W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])
y_conv = tf.nn.softmax(tf.matmul(h_fc1_drop, W_fc2) + b_fc2)
```

我们定义损失函数为 cross entropy，和之前一样，但是优化器使用 Adam，并给予一个比较小的学习速率 $1e-4$ 。

```
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y_conv),
                                             reduction_indices=[1]))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
```

再继续定义评测准确率的操作，这里和第 3 章、第 4 章一样。

```
correct_prediction = tf.equal(tf.argmax(y_conv, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

下面开始训练过程。首先依然是初始化所有参数，设置训练时 Dropout 的 keep_prob 比率为 0.5。然后使用大小为 50 的 mini-batch，共进行 20000 次训练迭代，参与训练的样本数量总共为 100 万。其中每 100 次训练，我们会对准确率进行一次评测（评测时 keep_prob 设为 1），用以实时监测模型的性能。

```
tf.global_variables_initializer().run()
for i in range(20000):
    batch = mnist.train.next_batch(50)
    if i%100 == 0:
        train_accuracy = accuracy.eval(feed_dict={x:batch[0], y_: batch[1],
                                                    keep_prob: 1.0})
        print("step %d, training accuracy %g"%(i, train_accuracy))
    train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})
```

全部训练完成后，我们在最终的测试集上进行全面的测试，得到整体的分类准确率。

```
print("test accuracy %g"%accuracy.eval(feed_dict={
    x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0}))
```

最后，这个 CNN 模型可以得到的准确率约为 99.2%，基本可以满足对手写数字识别准确率的要求。相比之前 MLP 的 2% 错误率，CNN 的错误率下降了大约 60%。这其中主要的性能提升都来自于更优秀的网络设计，即卷积网络对图像特征的提取和抽象能力。依靠卷积核的权值共享，CNN 的参数量并没有爆炸，降低计算量的同时也减轻了过拟合，因此整个模型的性能有较大的提升。本节我们只实现了一个简单的卷积神经网络，没有复杂的 Trick。接下来，我们将实现一个稍微复杂一些的卷积网络，而简单的 MNIST 数据集已经不适合用来评测其性能，我们将使用 CIFAR-10⁴² 数据集进行训练，这也是深度学习可以大幅领先其他模型的一个数据集。

5.3 TensorFlow 实现进阶的卷积网络

本节使用的数据集是 CIFAR-10，这是一个经典的数据集，包含 60000 张 32×32 的彩色图像，其中训练集 50000 张，测试集 10000 张。CIFAR-10 如同其名字，一共标注为 10 类，每一类图片 6000 张。这 10 类分别是 airplane、automobile、bird、cat、deer、dog、frog、horse、ship 和 truck，其中没有任何重叠的情况，比如 automobile 只包括小型汽车，truck

只包括卡车，也不会在一张图片中同时出现两类物体。它还有一个兄弟版本 CIFAR-100，其中标注了 100 类。这两个数据集是前面章节提到的深度学习之父 Geoffrey Hinton 和他的两名学生 Alex Krizhevsky 和 Vinod Nair 收集的，图片来源于 80 million tiny images⁴³ 这个数据集，Hinton 等人对其进行了筛选和标注。CIFAR-10 数据集非常通用，经常出现在各大会议的论文中用来进行性能对比，也曾出现在 Kaggle 竞赛而为大家所知。图 5-5 所示为这个数据集的一些示例。

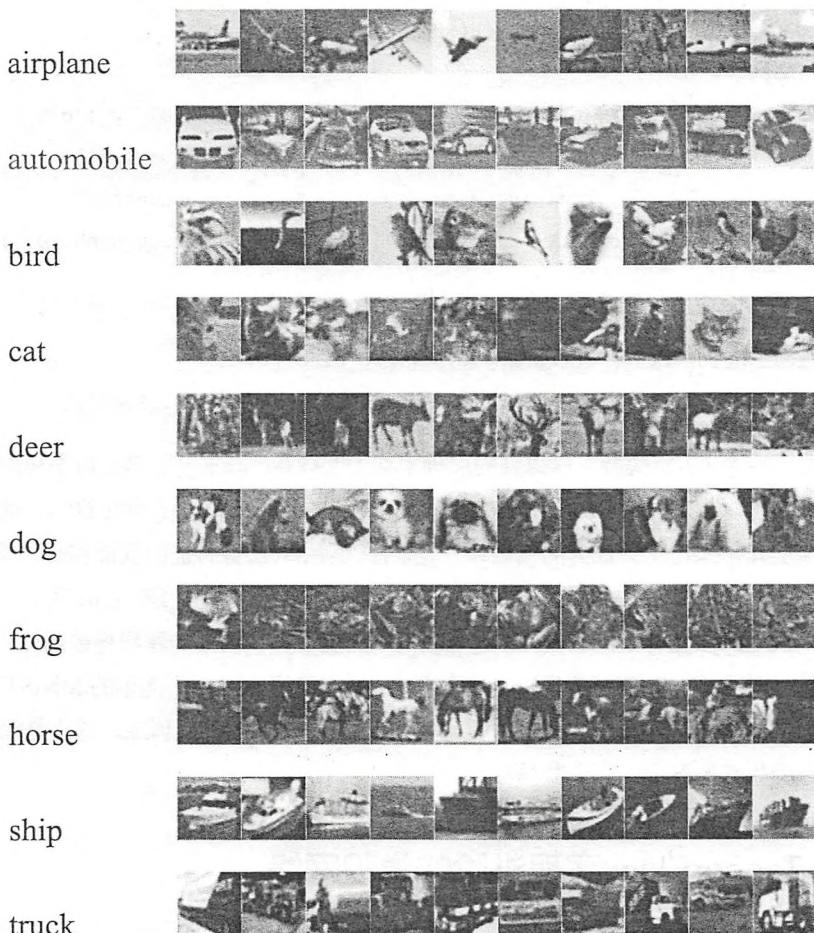


图 5-5 CIFAR-10 数据集示例

许多论文中都在这个数据集上进行了测试，目前 state-of-the-art 的工作已经可以达到 3.5% 的错误率了，但是需要训练很久，即使在 GPU 上也需要十几个小时。CIFAR-10 数

据集上详细的 Benchmark 和排名在 classification datasets results 上 (http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html)。据深度学习三巨头之一 LeCun 说，现有的卷积神经网络已经可以对 CIFAR-10 进行很好的学习，这个数据集的问题已经解决了。本节中实现的卷积神经网络没有那么复杂（根据 Alex 描述的 cuda-convnet 模型做了些许修改得到），在只使用 3000 个 batch（每个 batch 包含 128 个样本）时，可以达到 73% 左右的正确率。模型在 GTX 1080 单显卡上大概只需要几十秒的训练时间，如果在 CPU 上训练则会慢很多。如果使用 100k 个 batch，并结合学习速度的 decay（即每隔一段时间将学习速率下降一个比率），正确率最高可以到 86% 左右。模型中需要训练的参数约为 100 万个，而预测时需要进行的四则运算总量在 2000 万次左右。在这个卷积神经网络模型中，我们使用了一些新的技巧。

(1) 对 weights 进行了 L2 的正则化。

(2) 如图 5-6 所示，我们对图片进行了翻转、随机剪切等数据增强，制造了更多样本。

(3) 在每个卷积-最大池化层后面使用了 LRN 层，增强了模型的泛化能力。

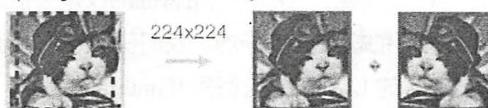
Data Augmentation:

a. No augmentation (= 1 image)



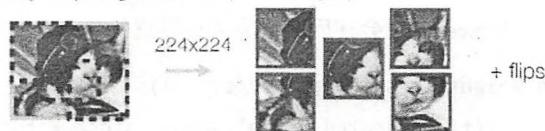
224x224

b. Flip augmentation (= 2 images)



224x224

c. Crop+Flip augmentation (= 10 images)



224x224

+ flips

图 5-6 数据增强示例（水平翻转，随机裁切）

我们首先下载 TensorFlow Models 库，以便使用其中提供 CIFAR-10 数据的类。

```
git clone https://github.com/tensorflow/models.git
cd models/tutorials/image/cifar10
```

然后我们载入一些常用库，比如 NumPy 和 time，并载入 TensorFlow Models 中自动下载、读取 CIFAR-10 数据的类。本节代码主要来自 TensorFlow 的开源实现⁴⁴。

```
import cifar10,cifar10_input
import tensorflow as tf
import numpy as np
import time
```

接着定义 batch_size、训练轮数 max_steps，以及下载 CIFAR-10 数据的默认路径。

```
max_steps = 3000
batch_size = 128
data_dir = '/tmp/cifar10_data/cifar-10-batches-bin'
```

这里定义初始化 weight 的函数，和之前一样依然使用 `tf.truncated_normal` 截断的正态分布来初始化权重。但是这里会给 weight 加一个 L2 的 loss，相当于做了一个 L2 的正则化处理。在机器学习中，不管是分类还是回归任务，都可能因特征过多而导致过拟合，一般可以通过减少特征或者惩罚不重要特征的权重来缓解这个问题。但是通常我们并不知道该惩罚哪些特征的权重，而正则化就是帮助我们惩罚特征权重的，即特征的权重也会成为模型的损失函数的一部分。可以理解为，为了使用某个特征，我们需要付出 loss 的代价，除非这个特征非常有效，否则就会被 loss 上的增加覆盖效果。这样我们就可以筛选出最有效的特征，减少特征权重防止过拟合。这也即是奥卡姆剃刀法则，越简单的东西越有效。一般来说，L1 正则会制造稀疏的特征，大部分无用特征的权重会被置为 0，而 L2 正则会让特征的权重不过大，使得特征的权重比较平均。我们使用 `wl` 控制 L2 loss 的大小，使用 `tf.nn.l2_loss` 函数计算 weight 的 L2 loss，再使用 `tf.multiply` 让 L2 loss 乘以 `wl`，得到最后的 weight loss。接着，我们使用 `tf.add_to_collection` 把 weight loss 统一存到一个 collection，这个 collection 名为“losses”，它会在后面计算神经网络的总体 loss 时被用上。

```
def variable_with_weight_loss(shape, stddev, wl):
    var = tf.Variable(tf.truncated_normal(shape, stddev=stddev))
    if wl is not None:
        weight_loss = tf.multiply(tf.nn.l2_loss(var), wl, name='weight_loss')
        tf.add_to_collection('losses', weight_loss)
    return var
```

下面使用 `cifar10` 类下载数据集，并解压、展开到其默认位置。

```
cifar10.maybe_download_and_extract()
```

再使用 cifar10_input 类中的 distorted_inputs 函数产生训练需要使用的数据，包括特征及其对应的 label，这里返回的是已经封装好的 tensor，每次执行都会生成一个 batch_size 的数量的样本。需要注意的是我们对数据进行了 Data Augmentation（数据增强）。具体的实现细节，读者可以查看 cifar10_input.distorted_inputs 函数，其中的数据增强操作包括随机的水平翻转（tf.image.random_flip_left_right）、随机剪切一块 24×24 大小的图片（tf.random_crop）、设置随机的亮度和对比度（tf.image.random_brightness、tf.image.random_contrast），以及对数据进行标准化 tf.image.per_image_whitening（对数据减去均值，除以方差，保证数据零均值，方差为 1）。通过这些操作，我们可以获得更多的样本（带噪声的），原来的一张图片样本可以变为多张图片，相当于扩大样本量，对提高准确率非常有帮助。需要注意的是，我们对图像进行数据增强的操作需要耗费大量 CPU 时间，因此 distorted_inputs 使用了 16 个独立的线程来加速任务，函数内部会产生线程池，在需要使用时会通过 TensorFlow queue 进行调度。

```
images_train, labels_train = cifar10_input.distorted_inputs(  
    data_dir=data_dir, batch_size=batch_size)
```

我们再使用 cifar10_input.inputs 函数生成测试数据，这里不需要进行太多处理，不需要对图片进行翻转或修改亮度、对比度，不过需要裁剪图片正中间的 24×24 大小的区块，并进行数据标准化操作。

```
images_test, labels_test = cifar10_input.inputs(eval_data=True,  
    data_dir=data_dir,  
    batch_size=batch_size)
```

这里创建输入数据的 placeholder，包括特征和 label。在设定 placeholder 的数据尺寸时需要注意，因为 batch_size 在之后定义网络结构时被用到了，所以数据尺寸中的第一个值即样本条数需要被预先设定，而不能像以前一样可以设为 None。而数据尺寸中的图片尺寸为 24×24，即是裁剪后的大小，而颜色通道数则设为 3，代表图片是彩色有 RGB 三条通道。

```
image_holder = tf.placeholder(tf.float32, [batch_size, 24, 24, 3])  
label_holder = tf.placeholder(tf.int32, [batch_size])
```

做好了准备工作，接下来开始创建第一个卷积层。先使用之前写好的

`variable_with_weight_loss` 函数创建卷积核的参数并进行初始化。第一个卷积层使用 5×5 的卷积核大小, 3 个颜色通道, 64 个卷积核, 同时设置 weight 初始化函数的标准差为 0.05。我们不对第一个卷积层的 weight 进行 L2 的正则, 因此 `wl` (`weight loss`) 这一项设为 0。下面使用 `tf.nn.conv2d` 函数对输入数据 `image_holder` 进行卷积操作, 这里的步长 `stride` 均设为 1, `padding` 模式为 `SAME`。把这层的 `bias` 全部初始化为 0, 再将卷积的结果加上 `bias`, 最后使用一个 `ReLU` 激活函数进行非线性化。在 `ReLU` 激活函数之后, 我们使用一个尺寸为 3×3 且步长为 2×2 的最大池化层处理数据, 注意这里最大池化的尺寸和步长不一致, 这样可以增加数据的丰富性。再之后, 我们使用 `tf.nn.lrn` 函数, 即 `LRN` 对结果进行处理。`LRN` 最早见于 Alex 那篇用 CNN 参加 ImageNet 比赛的论文, Alex 在论文中解释 `LRN` 层模仿了生物神经系统的“侧抑制”机制, 对局部神经元的活动创建竞争环境, 使得其中响应比较大的值变得相对更大, 并抑制其他反馈较小的神经元, 增强了模型的泛化能力。Alex 在 ImageNet 数据集上的实验表明, 使用 `LRN` 后 CNN 在 Top1 的错误率可以降低 1.4%, 因此在其经典的 AlexNet 中使用了 `LRN` 层。`LRN` 对 `ReLU` 这种没有上限边界的激活函数会比较有用, 因为它会从附近的多个卷积核的响应 (Response) 中挑选比较大的反馈, 但不适合 `Sigmoid` 这种有固定边界并且能抑制过大值的激活函数。

```
weight1 = variable_with_weight_loss(shape=[5, 5, 3, 64], stddev=5e-2,
                                     wl=0.0)
kernel1 = tf.nn.conv2d(image_holder, weight1, [1, 1, 1, 1], padding='SAME')
bias1 = tf.Variable(tf.constant(0.0, shape=[64]))
conv1 = tf.nn.relu(tf.nn.bias_add(kernel1, bias1))
pool1 = tf.nn.max_pool(conv1, ksize=[1, 3, 3, 1], strides=[1, 2, 2, 1],
                      padding='SAME')
norm1 = tf.nn.lrn(pool1, 4, bias=1.0, alpha=0.001 / 9.0, beta=0.75)
```

现在来创建第二个卷积层, 这里的步骤和第一步很像, 区别如下。上一层的卷积核数量为 64 (即输出 64 个通道), 所以本层卷积核尺寸的第三个维度即输入的通道数也需要调整为 64; 还有一个需要注意的地方是这里的 `bias` 值全部初始化为 0.1, 而不是 0。最后, 我们调换了最大池化层和 `LRN` 层的顺序, 先进行 `LRN` 层处理, 再使用最大池化层。

```
weight2 = variable_with_weight_loss(shape=[5, 5, 64, 64], stddev=5e-2,
                                     wl=0.0)
kernel2 = tf.nn.conv2d(norm1, weight2, [1, 1, 1, 1], padding='SAME')
bias2 = tf.Variable(tf.constant(0.1, shape=[64]))
```

```

conv2 = tf.nn.relu(tf.nn.bias_add(kernel2, bias2))
norm2 = tf.nn.lrn(conv2, 4, bias=1.0, alpha=0.001 / 9.0, beta=0.75)
pool2 = tf.nn.max_pool(norm2, ksize=[1, 3, 3, 1], strides=[1, 2, 2, 1],
                      padding='SAME')

```

在两个卷积层之后，将使用一个全连接层，这里需要先把前面两个卷积层的输出结果全部 flatten，使用 `tf.reshape` 函数将每个样本都变成一维向量。我们使用 `get_shape` 函数，获取数据扁平化之后的长度。接着使用 `variable_with_weight_loss` 函数对全连接层的 weight 进行初始化，这里隐含节点数为 384，正态分布的标准差设为 0.04，bias 的值也初始化为 0.1。需要注意的是我们希望这个全连接层不要过拟合，因此设了一个非零的 weight loss 值 0.04，让这一层的所有参数都被 L2 正则所约束。最后我们依然使用 ReLU 激活函数进行非线性化。

```

reshape = tf.reshape(pool2, [batch_size, -1])
dim = reshape.get_shape()[1].value
weight3 = variable_with_weight_loss(shape=[dim, 384], stddev=0.04, wl=0.004)
bias3 = tf.Variable(tf.constant(0.1, shape=[384]))
local3 = tf.nn.relu(tf.matmul(reshape, weight3) + bias3)

```

接下来的这个全连接层和前一层很像，只不过其隐含节点数下降了一半，只有 192 个，其他的超参数保持不变。

```

weight4 = variable_with_weight_loss(shape=[384, 192], stddev=0.04, wl=0.004)
bias4 = tf.Variable(tf.constant(0.1, shape=[192]))
local4 = tf.nn.relu(tf.matmul(local3, weight4) + bias4)

```

下面是最后一层，依然先创建这一层的 weight，其正态分布标准差设为上一个隐含层的节点数的倒数，并且不计入 L2 的正则。需要注意的是，这里不像之前那样使用 softmax 输出最后结果，这是因为我们把 softmax 的操作放在了计算 loss 的部分。我们不需要对 inference 的输出进行 softmax 处理就可以获得最终分类结果（直接比较 inference 输出的各类的数值大小即可），计算 softmax 主要是为了计算 loss，因此 softmax 操作整合到后面是比较合适的。

```

weight5 = variable_with_weight_loss(shape=[192, 10], stddev=1/192.0, wl=0.0)
bias5 = tf.Variable(tf.constant(0.0, shape=[10]))
logits = tf.add(tf.matmul(local4, weight5), bias5)

```

到这里就完成了整个网络 inference 的部分。梳理整个网络结构可以得到表 5-1。从上到下，依次是整个卷积神经网络从输入到输出的流程。可以观察到，其实设计 CNN 主要就是安排卷积层、池化层、全连接层的分布和顺序，以及其中超参数的设置、Trick 的使用等。设计性能良好的 CNN 是有一定规律可循的，但是想要针对某个问题设计最合适的网络结构，是需要大量实践摸索的。

表 5-1 卷积神经网络结构表

Layer 名称	描 述
conv1	卷积层和 ReLU 激活函数
pool1	最大池化
norm1	LRN
conv2	卷积层和 ReLU 激活函数
norm2	LRN
pool2	最大池化
local3	全连接层和 ReLU 激活函数
local4	全连接层和 ReLU 激活函数
logits	模型 Inference 的输出结果

完成了模型 inference 部分的构建，接下来计算 CNN 的 loss。这里依然使用 cross entropy，需要注意的是我们把 softmax 的计算和 cross entropy loss 的计算合在了一起，即 `tf.nn.sparse_softmax_cross_entropy_with_logits`。这里使用 `tf.reduce_mean` 对 cross entropy 计算均值，再使用 `tf.add_to_collection` 把 cross entropy 的 loss 添加到整体 losses 的 collection 中。最后，使用 `tf.add_n` 将整体 losses 的 collection 中的全部 loss 求和，得到最终的 loss，其中包括 cross entropy loss，还有后两个全连接层中 weight 的 L2 loss。

```
def loss(logits, labels):
    labels = tf.cast(labels, tf.int64)
    cross_entropy = tf.nn.sparse_softmax_cross_entropy_with_logits(
        logits=logits, labels=labels, name='cross_entropy_per_example')
    cross_entropy_mean = tf.reduce_mean(cross_entropy,
                                        name='cross_entropy')
    tf.add_to_collection('losses', cross_entropy_mean)

    return tf.add_n(tf.get_collection('losses'), name='total_loss')
```

接着将 logits 节点和 label_placeholder 传入 loss 函数获得最终的 loss。

```
loss = loss(logits, label_holder)
```

优化器依然选择 Adam Optimizer，学习速率设为 1e-3。

```
train_op = tf.train.AdamOptimizer(1e-3).minimize(loss)
```

使用 tf.nn.in_top_k 函数求输出结果中 top k 的准确率，默认使用 top 1，也就是输出分数最高的那一类的准确率。

```
top_k_op = tf.nn.in_top_k(logits, label_holder, 1)
```

使用 tf.InteractiveSession 创建默认的 session，接着初始化全部模型参数。

```
sess = tf.InteractiveSession()
tf.global_variables_initializer().run()
```

这一步是启动前面提到的图片数据增强的线程队列，这里一共使用了 16 个线程来进行加速。注意，如果这里不启动线程，那么后续的 inference 及训练的操作都是无法开始的。

```
tf.train.start_queue_runners()
```

现在正式开始训练。在每一个 step 的训练过程中，我们需要先使用 session 的 run 方法执行 images_train、labels_train 的计算，获得一个 batch 的训练数据，再将这个 batch 的数据传入 train_op 和 loss 的计算。我们记录每一个 step 花费的时间，每隔 10 个 step 会计算并展示当前的 loss、每秒钟能训练的样本数量，以及训练一个 batch 数据所花费的时间，这样就可以比较方便地监控整个训练过程。在 GTX 1080 上，每秒钟可以训练大约 1800 个样本，如果 batch_size 为 128，则每个 batch 大约需要 0.066s。损失 loss 在一开始大约为 4.6，在经过了 3000 步训练后会下降到 1.0 附近。

```
for step in range(max_steps):
    start_time = time.time()
    image_batch, label_batch = sess.run([images_train, labels_train])
    _, loss_value = sess.run([train_op, loss],
                           feed_dict={image_holder: image_batch, label_holder: label_batch})
    duration = time.time() - start_time
```

```

if step % 10 == 0:
    examples_per_sec = batch_size / duration
    sec_per_batch = float(duration)

    format_str = ('step %d, loss=% .2f (%.1f examples/sec; %.3f sec/batch)')
    print(format_str % (step, loss_value, examples_per_sec, sec_per_batch))

```

接下来评测模型在测试集上的准确率。测试集一共有 10000 个样本，但是需要注意的是，我们依然要像训练时那样使用固定的 batch_size，然后一个 batch 一个 batch 地输入测试数据。我们先计算一共要多少个 batch 才能将全部样本评测完。同时，在每一个 step 中使用 session 的 run 方法获取 images_test、labels_test 的 batch，再执行 top_k_op 计算模型在这个 batch 的 top 1 上预测正确的样本数。最后汇总所有预测正确的结果，求得全部测试样本中预测正确的数量。

```

num_examples = 10000
import math
num_iter = int(math.ceil(num_examples / batch_size))
true_count = 0
total_sample_count = num_iter * batch_size
step = 0
while step < num_iter:
    image_batch, label_batch = sess.run([images_test, labels_test])
    predictions = sess.run([top_k_op], feed_dict={image_holder: image_batch,
                                                label_holder: label_batch})
    true_count += np.sum(predictions)
    step += 1

```

最后将准确率的评测结果计算并打印出来。

```

precision = true_count / total_sample_count
print('precision @ 1 = %.3f' % precision)

```

最终，在 CIFAR-10 数据集上，通过一个短时间小迭代次数的训练，可以达到大致 73% 的准确率。持续增加 max_steps，可以期望准确率逐渐增加。如果 max_steps 比较大，则推荐使用学习速率衰减（decay）的 SGD 进行训练，这样训练过程中能达到的准确率峰值

会比较高，大致接近 86%。而其中 L2 正则及 LRN 层的使用都对模型准确率有提升作用，他们都可以从某些方面提升模型的泛化性。

数据增强（Data Augmentation）在我们的训练中作用很大，它可以给单幅图增加多个副本，提高图片的利用率，防止对某一张图片结构的学习过拟合。这刚好是利用了图片数据本身的性质，图片的冗余信息量比较大，因此可以制造不同的噪声并让图片依然可以被识别出来。如果神经网络可以克服这些噪声并准确识别，那么它的泛化性必然会很好。数据增强大大增加了样本量，而数据量的大小恰恰是深度学习最看重的，深度学习可以在图像识别上领先其他算法的一大因素就是它对海量数据的利用效率非常高。用其他算法，可能在数据量大到一定程度时，准确率就不再上升了，而深度学习只要提供足够多的样本，准确率基本可以持续提升，所以说它是最适合大数据的算法。如图 5-6 所示，传统的机器学习算法在获取了一定量的数据后，准确率上升曲线就接近瓶颈，而神经网络则可以持续上升到更高的准确率才接近瓶颈。规模越大越复杂的神经网络模型，可以达到的准确率水平越高，但是也相应地需要更多的数据才能训练好，在数据量小时反而容易过拟合。我们可以看到 Large NN 在数据量小的时候，并不比常规算法好，直到数据量持续扩大才慢慢超越了常规算法、Small NN 和 Medium NN，并在最后达到了一个非常高的准确率。根据 Alex 在 cuda-convnet 上的测试结果，如果不对 CIFAR-10 数据使用数据增强，那么错误率最低可以下降到 17%；使用数据增强后，错误率可以下降到 11% 左右，模型性能的提升非常显著。

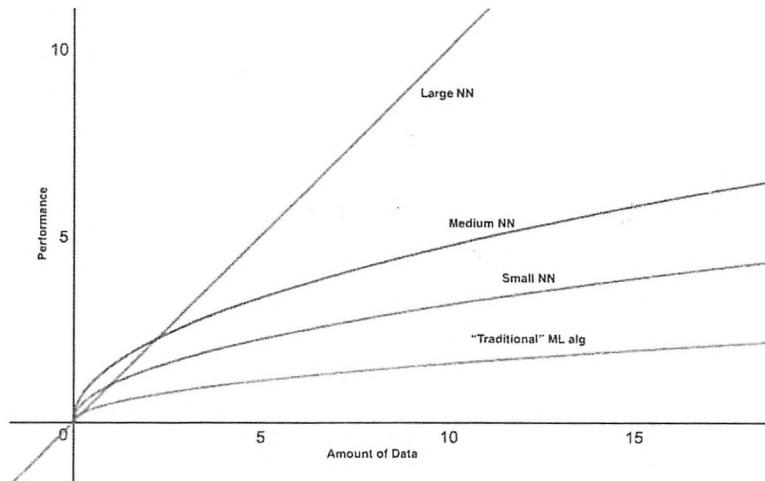
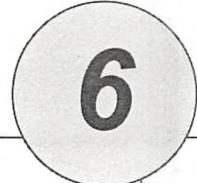


图 5-6 传统机器学习算法和深度学习在不同数据量下的表现

从本章的例子中可以发现，卷积层一般需要和一个池化层连接，卷积加池化的组合目前已经是做图像识别时的一个标准组件了。卷积网络最后的几个全连接层的作用是输出分类结果，前面的卷积层主要做特征提取的工作，直到最后的全连接层才开始对特征进行组合匹配，并进行分类。卷积层的训练相对于全连接层更复杂，训练全连接层基本是进行一些矩阵乘法运算，而目前卷积层的训练基本依赖于 cuDNN 的实现（另有 nervana 公司的 neon 也占有一席之地）。其中的算法相对复杂，有些方法（比如 Facebook 开源的算法）还会涉及傅里叶变换。同时，卷积层的使用有很多 Trick，除了本章提到的方法，实际上有很多方法可以防止 CNN 过拟合，加快收敛速度或者提高泛化性，这些会在后续章节中讲解。



6

TensorFlow 实现经典卷 积神经网络

本章将介绍 4 种经典的卷积神经网络，分别是 AlexNet⁴⁵、VGGNet⁴⁶、Google Inception Net⁴⁷ 和 ResNet⁴⁸，这 4 种网络依照出现的先后顺序排列，深度和复杂度也依次递进。它们分别获得了 ILSVRC (ImageNet Large Scale Visual Recognition Challenge)⁴⁹ 比赛分类项目的 2012 年冠军 (top-5 错误率 16.4%，使用额外数据可达到 15.3%，8 层神经网络)、2014 年亚军 (top-5 错误率 7.3%，19 层神经网络)，2014 年冠军 (top-5 错误率 6.7%，22 层神经网络) 和 2015 年的冠军 (top-5 错误率 3.57%，152 层神经网络)。这 4 个经典的网络都在各自的年代率先使用了很多先进的卷积神经网络结构，对卷积网络乃至深度学习有非常大的推动作用，也象征了卷积神经网络在 2012—2015 这四年间的快速发展。如图 6-1 所示，ILSVRC 的 top-5 错误率在最近几年取得重大突破，而主要的突破点都是在深度学习和卷积神经网络，成绩的大幅提升几乎都伴随着卷积神经网络的层数加深。而传统机器学习算法目前在 ILSVRC 上已经难以追上深度学习的步伐了，以至于逐渐被称为浅层学习 (Shallow Learning)。目前在 ImageNet⁵⁰ 数据集上人眼能达到的错误率大概在 5.1%，这还是经过了大量训练的专家能达到的成绩，一般人要区分 1000 种类型的图片是比较困难的。而 ILSVRC 2015 年冠军——152 层 ResNet 的成绩达到错误率 3.57%，已经超过了

人眼，这说明卷积神经网络已经基本解决了 ImageNet 数据集上的图片分类问题。

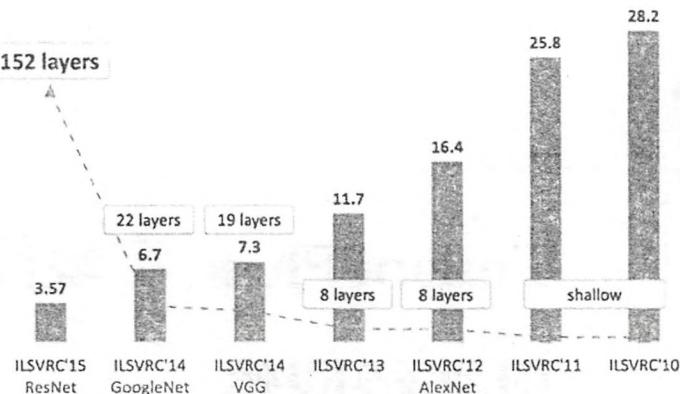


图 6-1 历届 ILSVRC 比赛代表性模型的成绩及其神经网络深度

前面提到的计算机视觉比赛 ILSVRC 使用的数据都来自 ImageNet，如图 6-2 所示。ImageNet 项目于 2007 年由斯坦福大学华人教授李飞飞创办，目标是收集大量带有标注信息的图片数据供计算机视觉模型训练。ImageNet 拥有 1500 万张标注过的高清图片，总共拥有 22000 类，其中约有 100 万张标注了图片中主要物体的定位边框。ImageNet 项目最早的灵感来自于人类通过视觉学习世界的方式，如果假定儿童的眼睛是生物照相机，他们平均每 200ms 就拍照一次（眼球转动一次的平均时间），那么 3 岁大时孩子就已经看过了上亿张真实世界的照片，可以算得上是一个非常大的数据集。ImageNet 项目下载了互联网上近 10 亿张图片，使用亚马逊的土耳其机器人平台实现众包的标注过程，有来自世界上 167 个国家的近 5 万名工作者帮忙一起筛选、标注。

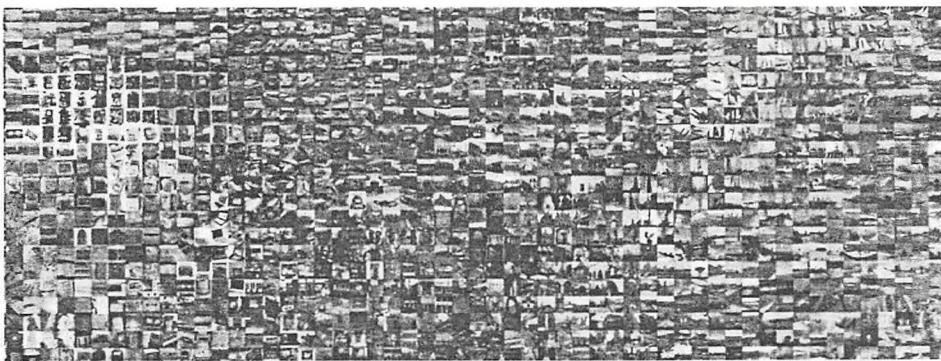


图 6-2 ImageNet 数据集图片示例

每年度的 ILSVRC 比赛数据集中大概拥有 120 万张图片，以及 1000 类的标注，是 ImageNet 全部数据的一个子集。比赛一般采用 top-5 和 top-1 分类错误率作为模型性能的评测指标，图 6-3 所示为 AlexNet 识别 ILSVRC 数据集中图片的情况，每张图片下面是分类预测得分最高的 5 个分类及其分值。

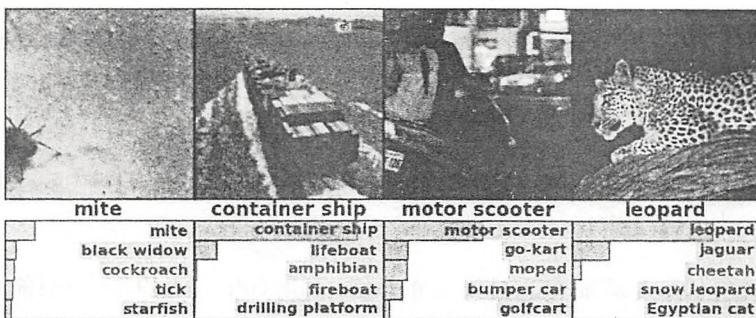


图 6-3 AlexNet 识别 ILSVRC 数据集的 top-5 分类

6.1 TensorFlow 实现 AlexNet

2012 年，Hinton 的学生 Alex Krizhevsky 提出了深度卷积神经网络模型 AlexNet，它可以算是 LeNet 的一种更深更宽的版本。AlexNet 中包含了几个比较新的技术点，也首次在 CNN 中成功应用了 ReLU、Dropout 和 LRN 等 Trick。同时 AlexNet 也使用了 GPU 进行运算加速，作者开源了他们在 GPU 上训练卷积神经网络的 CUDA 代码。AlexNet 包含了 6 亿 3000 万个连接，6000 万个参数和 65 万个神经元，拥有 5 个卷积层，其中 3 个卷积层后面连接了最大池化层，最后还有 3 个全连接层。AlexNet 以显著的优势赢得了竞争激烈的 ILSVRC 2012 比赛，top-5 的错误率降低至了 16.4%，相比第二名的成绩 26.2% 错误率有了巨大的提升。AlexNet 可以说是神经网络在低谷期后的第一次发声，确立了深度学习（深度卷积网络）在计算机视觉的统治地位，同时也推动了深度学习在语音识别、自然语言处理、强化学习等领域的拓展。

AlexNet 将 LeNet 的思想发扬光大，把 CNN 的基本原理应用到了很深很宽的网络中。AlexNet 主要使用到的新技术点如下。

(1) 成功使用 ReLU 作为 CNN 的激活函数，并验证其效果在较深的网络超过了 Sigmoid，成功解决了 Sigmoid 在网络较深时的梯度弥散问题。虽然 ReLU 激活函数在很

久之前就被提出了，但是直到 AlexNet 的出现才将其发扬光大。

(2) 训练时使用 Dropout 随机忽略一部分神经元，以避免模型过拟合。Dropout 虽有单独的论文论述，但是 AlexNet 将其实用化，通过实践证实了它的效果。在 AlexNet 中主要是最后几个全连接层使用了 Dropout。

(3) 在 CNN 中使用重叠的最大池化。此前 CNN 中普遍使用平均池化，AlexNet 全部使用最大池化，避免平均池化的模糊化效果。并且 AlexNet 中提出让步长比池化核的尺寸小，这样池化层的输出之间会有重叠和覆盖，提升了特征的丰富性。

(4) 提出了 LRN 层，对局部神经元的活动创建竞争机制，使得其中响应比较大的值变得相对更大，并抑制其他反馈较小的神经元，增强了模型的泛化能力。

(5) 使用 CUDA 加速深度卷积网络的训练，利用 GPU 强大的并行计算能力，处理神经网络训练时大量的矩阵运算。AlexNet 使用了两块 GTX 580 GPU 进行训练，单个 GTX 580 只有 3GB 显存，这限制了可训练的网络的最大规模。因此作者将 AlexNet 分布在两个 GPU 上，在每个 GPU 的显存中储存一半的神经元的参数。因为 GPU 之间通信方便，可以互相访问显存，而不需要通过主机内存，所以同时使用多块 GPU 也是非常高效的。同时，AlexNet 的设计让 GPU 之间的通信只在网络的某些层进行，控制了通信的性能损耗。

(6) 数据增强，随机地从 256×256 的原始图像中截取 224×224 大小的区域（以及水平翻转的镜像），相当于增加了 $(256-224)^2 \times 2 = 2048$ 倍的数据量。如果没有数据增强，仅靠原始的数据量，参数众多的 CNN 会陷入过拟合中，使用了数据增强后可以大大减轻过拟合，提升泛化能力。进行预测时，则是取图片的四个角加中间共 5 个位置，并进行左右翻转，一共获得 10 张图片，对他们进行预测并对 10 次结果求均值。同时，AlexNet 论文中提到了会对图像的 RGB 数据进行 PCA 处理，并对主成分做一个标准差为 0.1 的高斯扰动，增加一些噪声，这个 Trick 可以让错误率再下降 1%。

整个 AlexNet 有 8 个需要训练参数的层（不包括池化层和 LRN 层），前 5 层为卷积层，后 3 层为全连接层，如图 6-4 所示。AlexNet 最后一层是有 1000 类输出的 Softmax 层用作分类。LRN 层出现在第 1 个及第 2 个卷积层后，而最大池化层出现在两个 LRN 层及最后一个卷积层后。ReLU 激活函数则应用在这 8 层每一层的后面。因为 AlexNet 训练时使用了两块 GPU，因此这个结构图中不少组件都被拆为了两部分。现在我们 GPU 的显存可以放下全部模型参数，因此只考虑一块 GPU 的情况即可。

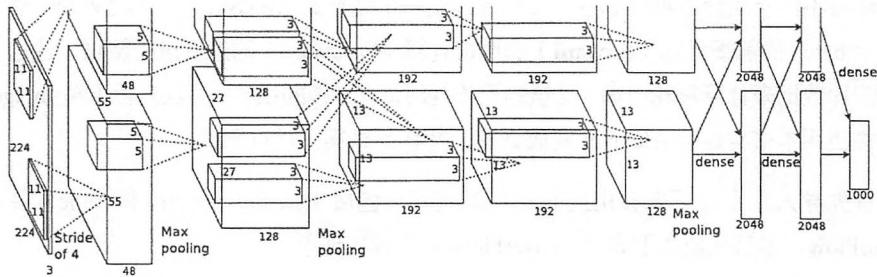


图 6-4 AlexNet 的网络结构

AlexNet 每层的超参数如图 6-5 所示。其中输入的图片尺寸为 224×224 ，第一个卷积层使用了较大的卷积核尺寸 11×11 ，步长为 4，有 96 个卷积核；紧接着一个 LRN 层；然后是一个 3×3 的最大池化层，步长为 2。这之后的卷积核尺寸都比较小，都是 5×5 或者 3×3 的大小，并且步长都为 1，即会扫描全图所有像素；而最大池化层依然保持为 3×3 ，并且步长为 2。我们可以发现一个比较有意思的现象，在前几个卷积层，虽然计算量很大，但参数量很小，都在 1M 左右甚至更小，只占 AlexNet 总参数量的很小一部分。这就是卷积层有用的地方，可以通过较小的参数量提取有效的特征。而如果前几层直接使用全连接层，那么参数量和计算量将成为天文数字。虽然每一个卷积层占整个网络的参数量的 1% 都不到，但是如果去掉任何一个卷积层，都会使网络的分类性能大幅地下降。

params	AlexNet	FLOPs
4M	FC 1000	4M
16M	FC 4096 / ReLU	16M
37M	FC 4096 / ReLU	37M
442K	Max Pool 3x3s2	
1.3M	Conv 3x3s1, 256 / ReLU	74M
884K	Conv 3x3s1, 384 / ReLU	112M
307K	Conv 3x3s1, 384 / ReLU	149M
35K	Max Pool 3x3s2	
307K	Local Response Norm	
35K	Conv 5x5s1, 256 / ReLU	223M
35K	Max Pool 3x3s2	
35K	Local Response Norm	
35K	Conv 11x11s4, 96 / ReLU	105M

图 6-5 AlexNet 每层的超参数及参数数量

因为使用 ImageNet 数据集训练一个完整的 AlexNet 耗时非常长，因此本节中 AlexNet

的实现将不涉及实际数据的训练。我们会建立一个完整的 AlexNet 卷积神经网络，然后对它每个 batch 的前馈计算（forward）和反馈计算（backward）的速度进行测试。这里使用随机图片数据来计算每轮前馈、反馈的平均耗时。有兴趣的读者，可以自行下载 ImageNet 数据并使用本书构建的 AlexNet 完成训练，并在测试集上进行测试。

首先导入几个接下来会用到的几个系统库，包括 `datetime`、`math` 和 `time`，并载入 TensorFlow。本节代码主要来自 TensorFlow 的开源实现⁵¹。

```
from datetime import datetime
import math
import time
import tensorflow as tf
```

这里设置 `batch_size` 为 32，`num_batches` 为 100，即总共测试 100 个 batch 的数据。

```
batch_size=32
num_batches=100
```

定义一个用来显示网络每一层结构的函数 `print_activations`，展示每一个卷积层或池化层输出 tensor 的尺寸。这个函数接受一个 tensor 作为输入，并显示其名称 (`t.op.name`) 和 tensor 尺寸 (`t.get_shape().as_list()`)。

```
def print_activations(t):
    print(t.op.name, ' ', t.get_shape().as_list())
```

接下来设计 AlexNet 的网络结构。我们先定义函数 `inference`，它接受 `images` 作为输入，返回最后一层 `pool5`（第 5 个池化层）及 `parameters`（AlexNet 中所有需要训练的模型参数）。这个 `inference` 函数将会很大，包括多个卷积和池化层，因此下面将拆为几个小段分别讲解。

首先是第一个卷积层 `conv1`，这里使用 TensorFlow 中的 `name_scope`，通过 `tf.name_scope('conv1') as scope` 可以将 scope 内生成的 Variable 自动命名为 `conv1/xxx`，便于区分不同卷积层之间的组件。然后定义第一个卷积层，和之前一样使用 `tf.truncated_normal` 截断的正态分布函数（标准差为 0.1）初始化卷积核的参数 `kernel`。卷积核尺寸为 11×11 ，颜色通道为 3，卷积核数量为 64。准备好了 `kernel`，再使用 `tf.nn.conv2d` 对输入 `images` 完成卷积操作，我们将 `strides` 步长设置为 4×4 （即在图片上每 4×4 区域只取样一次，横向间隔是 4，纵向间隔也为 4，每次取样的卷积核大小都为 11×11 ），`padding`

模式设为 SAME。将卷积层的 biases 全部初始化为 0, 再使用 tf.nn.bias_add 将 conv 和 biases 加起来, 并使用激活函数 tf.nn.relu 对结果进行非线性处理。最后使用 print_activations 将这一层最后输出的 tensor conv1 的结构打印出来, 并将这一层可训练的参数 kernel、biases 添加到 parameters 中。

```
def inference(images):
    parameters = []

    with tf.name_scope('conv1') as scope:
        kernel = tf.Variable(tf.truncated_normal([11, 11, 3, 64],
                                                dtype=tf.float32, stddev=1e-1), name='weights')
        conv = tf.nn.conv2d(images, kernel, [1, 4, 4, 1], padding= 'SAME')
        biases = tf.Variable(tf.constant(0.0, shape=[64], dtype=tf.float32),
                             trainable=True, name='biases')
        bias = tf.nn.bias_add(conv, biases)
        conv1 = tf.nn.relu(bias, name=scope)
        print_activations(conv1)
        parameters += [kernel, biases]
```

在第 1 个卷积层后再添加 LRN 层和最大池化层。先使用 tf.nn.lrn 对前面输出的 tensor conv1 进行 LRN 处理, 这里使用的 depth_radius 设为 4, bias 设为 1, alpha 为 0.001/9, beta 为 0.75, 基本都是 AlexNet 的论文中的推荐值。不过目前除了 AlexNet, 其他经典的卷积神经网络模型基本都放弃了 LRN (主要是效果不明显), 而我们使用 LRN 也会让前馈、反馈的速度大大下降 (整体速度降到 1/3), 读者可以自主选择是否使用 LRN。下面使用 tf.nn.max_pool 对前面的输出 lrn1 进行最大池化处理, 这里的池化尺寸为 3×3, 即将 3×3 大小的像素块降为 1×1 的像素, 取样的步长为 2×2, padding 模式设为 VALID, 即取样时不能超过边框, 不像 SAME 模式那样可以填充边界外的点。最后将输出结果 pool1 的结构打印出来。

```
lrn1 = tf.nn.lrn(conv1, 4, bias=1.0, alpha=0.001/9, beta=0.75, name='lrn1')
pool1 = tf.nn.max_pool(lrn1, ksize=[1, 3, 3, 1], strides=[1, 2, 2, 1],
                      padding='VALID', name='pool1')
print_activations(pool1)
```

接下来设计第 2 个卷积层, 大部分步骤和第 1 个卷积层相同, 只有几个参数不同。主

要区别在于我们的卷积核尺寸是 5×5 , 输入通道数(即上一层输出通道数,也就是上一层卷积核数量)为 64, 卷积核数量为 192。同时, 卷积的步长也全部设为 1, 即扫描全图像素。

```
with tf.name_scope('conv2') as scope:
    kernel = tf.Variable(tf.truncated_normal([5, 5, 64, 192],
                                              dtype=tf.float32, stddev=1e-1), name='weights')
    conv = tf.nn.conv2d(pool1, kernel, [1, 1, 1, 1], padding='SAME')
    biases = tf.Variable(tf.constant(0.0, shape=[192]),
                         dtype=tf.float32, trainable=True, name='biases')
    bias = tf.nn.bias_add(conv, biases)
    conv2 = tf.nn.relu(bias, name=scope)
    parameters += [kernel, biases]
print_activations(conv2)
```

接下来对第 2 个卷积层的输出 conv2 进行处理, 同样是先做 LRN 处理, 再进行最大池化处理, 参数和之前完全一样, 这里就不再赘述了。

```
lrn2 = tf.nn.lrn(conv2, 4, bias=1.0, alpha=0.001/9, beta=0.75, name='lrn2')
pool2 = tf.nn.max_pool(lrn2, ksize=[1, 3, 3, 1], strides=[1, 2, 2, 1],
                      padding='VALID', name='pool2')
print_activations(pool2)
```

下面创建第 3 个卷积层, 基本结构和前面两个类似, 也只是参数不同。这一层的卷积核尺寸为 3×3 , 输入的通道数为 192, 卷积核数量继续扩大为 384, 同时卷积的步长全部为 1, 其他地方和前面保持一致。

```
with tf.name_scope('conv3') as scope:
    kernel = tf.Variable(tf.truncated_normal([3, 3, 192, 384],
                                              dtype=tf.float32, stddev=1e-1), name='weights')
    conv = tf.nn.conv2d(pool2, kernel, [1, 1, 1, 1], padding='SAME')
    biases = tf.Variable(tf.constant(0.0, shape=[384]),
                         dtype=tf.float32, trainable=True, name='biases')
    bias = tf.nn.bias_add(conv, biases)
    conv3 = tf.nn.relu(bias, name=scope)
    parameters += [kernel, biases]
```

```
print_activations(conv3)
```

第4个卷积层和之前也类似，这一层的卷积核尺寸为 3×3 ，输入通道数为384，但是卷积核数量降为256。

```
with tf.name_scope('conv4') as scope:
    kernel = tf.Variable(tf.truncated_normal([3, 3, 384, 256],
                                              dtype=tf.float32, stddev=1e-1), name='weights')
    conv = tf.nn.conv2d(conv3, kernel, [1, 1, 1, 1], padding='SAME')
    biases = tf.Variable(tf.constant(0.0, shape=[256]),
                         dtype=tf.float32, trainable=True, name='biases')
    bias = tf.nn.bias_add(conv, biases)
    conv4 = tf.nn.relu(bias, name=scope)
    parameters += [kernel, biases]
    print_activations(conv4)
```

最后的第5个卷积层同样是 3×3 大小的卷积核，输入通道数为256，卷积核数量也为256。

```
with tf.name_scope('conv5') as scope:
    kernel = tf.Variable(tf.truncated_normal([3, 3, 256, 256],
                                              dtype=tf.float32, stddev=1e-1), name='weights')
    conv = tf.nn.conv2d(conv4, kernel, [1, 1, 1, 1], padding='SAME')
    biases = tf.Variable(tf.constant(0.0, shape=[256]),
                         dtype=tf.float32, trainable=True, name='biases')
    bias = tf.nn.bias_add(conv, biases)
    conv5 = tf.nn.relu(bias, name=scope)
    parameters += [kernel, biases]
    print_activations(conv5)
```

在第5个卷积层之后，还有一个最大池化层，这个池化层和前两个卷积层后的池化层一致，最后我们返回这个池化层的输出pool5。至此，inference函数就完成了，它可以创建AlexNet的卷积部分。在正式使用AlexNet来训练或预测时，还需要添加3个全连接层，隐含节点数分别为4096、4096和1000。由于最后3个全连接层的计算量很小，就没放到计算速度评测中，他们对计算耗时的影响非常小。读者在正式使用AlexNet时需要自行添

加这 3 个全连接层，全连接层在 TensorFlow 中的实现方法在第 4 章已经讲解过，这里不再赘述。

```
pool5 = tf.nn.max_pool(conv5, ksize=[1, 3, 3, 1], strides=[1, 2, 2, 1],
                       padding='VALID', name='pool5')
print_activations(pool5)

return pool5, parameters
```

接下来实现一个评估 AlexNet 每轮计算时间的函数 time_tensorflow_run。这个函数的第一个输入是 TensorFlow 的 Session，第二个变量是需要评测的运算算子，第三个变量是测试的名称。先定义预热轮数 num_steps_burn_in=10，它的作用是给程序热身，头几轮迭代有显存加载、cache 命中等问题因此可以跳过，我们只考量 10 轮迭代之后的计算时间。同时，也记录总时间 total_duration 和平方和 total_duration_squared 用以计算方差。

```
def time_tensorflow_run(session, target, info_string):
    num_steps_burn_in = 10
    total_duration = 0.0
    total_duration_squared = 0.0
```

我们进行 num_batches+num_steps_burn_in 次迭代计算，使用 time.time() 记录时间，每次迭代通过 session.run(target) 执行。在初始热身的 num_steps_burn_in 次迭代后，每 10 轮迭代显示当前迭代所需要的时间。同时每轮将 total_duration 和 total_duration_squared 累加，以便后面计算每轮耗时的均值和标准差。

```
for i in range(num_batches + num_steps_burn_in):
    start_time = time.time()
    _ = session.run(target)
    duration = time.time() - start_time
    if i >= num_steps_burn_in:
        if not i % 10:
            print('%s: step %d, duration = %.3f' %
                  (datetime.now(), i - num_steps_burn_in, duration))
    total_duration += duration
    total_duration_squared += duration * duration
```

在循环结束后，计算每轮迭代的平均耗时 mn 和标准差 sd，最后将结果显示出来。这样就完成了计算每轮迭代耗时的评测函数 time_tensorflow_run。

```
mn = total_duration / num_batches
vr = total_duration_squared / num_batches - mn * mn
sd = math.sqrt(vr)
print('%s: %s across %d steps, %.3f +/- %.3f sec / batch' %
      (datetime.now(), info_string, num_batches, mn, sd))
```

接下来是主函数 run_benchmark。首先使用 with tf.Graph().as_default() 定义默认的 Graph 方便后面使用。如前面所说，我们并不使用 ImageNet 数据集来训练，只使用随机图片数据测试前馈和反馈计算的耗时。我们使用 tf.random_normal 函数构造正态分布（标准差为 0.1）的随机 tensor，第一个维度是 batch_size，即每轮迭代的样本数，第二个和第三个维度是图片的尺寸 image_size=224，第四个维度是图片的颜色通道数。接下来，使用前面定义的 inference 函数构建整个 AlexNet 网络，得到最后一个池化层的输出 pool5 和网络中需要训练的参数的集合 parameters。接下来，我们使用 tf.Session() 创建新的 Session 并通过 tf.global_variables_initializer() 初始化所有参数。

```
def run_benchmark():
    with tf.Graph().as_default():
        image_size = 224
        images = tf.Variable(tf.random_normal([batch_size,
                                              image_size,
                                              image_size, 3],
                                              dtype=tf.float32,
                                              stddev=1e-1))

        pool5, parameters = inference(images)

        init = tf.global_variables_initializer()
        sess = tf.Session()
        sess.run(init)
```

下面进行 AlexNet 的 forward 计算的评测，这里直接使用 time_tensorflow_run 统计运算时间，传入的 target 就是 pool5，即卷积网络最后一个池化层的输出。然后进行 backward

即训练过程的评测，这里和 forward 计算有些不同，我们需要给最后的输出 pool5 设置一个优化目标 loss。我们使用 `tf.nn.l2_loss` 计算 pool5 的 loss，再使用 `tf.gradients` 求相对于 loss 的所有模型参数的梯度，这样就模拟了一个训练的过程。当然，训练时还有一个根据梯度更新参数的过程，不过这个计算量很小，就不统计在评测程序里了。最后我们使用 `time_tensorflow_run` 统计 backward 的运算时间，这里的 target 就是求整个网络梯度 `grad` 的操作。

```
time_tensorflow_run(sess, pool5, "Forward")

objective = tf.nn.l2_loss(pool5)
grad = tf.gradients(objective, parameters)
time_tensorflow_run(sess, grad, "Forward-backward")
```

最后执行主函数。

```
run_benchmark()
```

程序显示的结果有三段，首先是 AlexNet 的网络结构，可以看到我们定义的 5 个卷积层中第 1 个、第 2 个和第 5 个卷积层后面还连接着池化层，另外每一层输出 tensor 的尺寸也显示出来了。

```
conv1 [32, 56, 56, 64]
pool1 [32, 27, 27, 64]
conv2 [32, 27, 27, 192]
pool2 [32, 13, 13, 192]
conv3 [32, 13, 13, 384]
conv4 [32, 13, 13, 256]
conv5 [32, 13, 13, 256]
pool5 [32, 6, 6, 256]
```

然后显示的是 forward 计算的时间。我们使用的 GPU 是 GTX 1080，软件环境包括 CUDA 8.0 和 cuDNN 5.1。在有 LRN 层时每轮迭代时间大约为 0.026s；去除 LRN 层时每轮迭代时间大约为 0.007s，运算时间有了大幅缩减，大约快了 3 倍多。因为 LRN 层对最终准确率的影响不是很大，所以读者可以自行考虑是否使用 LRN。

```
2016-12-10 21:08:31.851750: step 0, duration = 0.026
2016-12-10 21:08:32.109889: step 10, duration = 0.026
```

```
2016-12-10 21:08:32.367558: step 20, duration = 0.026
2016-12-10 21:08:32.625277: step 30, duration = 0.026
2016-12-10 21:08:32.884085: step 40, duration = 0.026
2016-12-10 21:08:33.141951: step 50, duration = 0.026
2016-12-10 21:08:33.400239: step 60, duration = 0.026
2016-12-10 21:08:33.658713: step 70, duration = 0.026
2016-12-10 21:08:33.916780: step 80, duration = 0.026
2016-12-10 21:08:34.174840: step 90, duration = 0.026
2016-12-10 21:08:34.407022: Forward across 100 steps, 0.026 +/- 0.000 sec /
batch
```

然后是显示的 backward 运算的时间。在使用 LRN 层时，每轮的迭代时间为 0.078s；在去除 LRN 层后，每轮迭代时间约为 0.025s，速度也快了 3 倍多。另外可以发现不论是否有 LRN 层，我们 backward 运算的耗时大约是 forward 耗时的三倍。

```
2016-12-10 21:08:35.447963: step 0, duration = 0.078
2016-12-10 21:08:36.225855: step 10, duration = 0.078
2016-12-10 21:08:37.002277: step 20, duration = 0.077
2016-12-10 21:08:37.777730: step 30, duration = 0.078
2016-12-10 21:08:38.555231: step 40, duration = 0.078
2016-12-10 21:08:39.333113: step 50, duration = 0.077
2016-12-10 21:08:40.110716: step 60, duration = 0.078
2016-12-10 21:08:40.887492: step 70, duration = 0.078
2016-12-10 21:08:41.664907: step 80, duration = 0.078
2016-12-10 21:08:42.441733: step 90, duration = 0.078
2016-12-10 21:08:43.139532: Forward-backward across 100 steps, 0.078 +/- 0.0
00 sec / batch
```

CNN 的训练过程(即 backward 计算)通常都比较耗时，而且不像预测过程(即 forward 计算)，训练通常需要过很多遍数据，进行大量的迭代。因此应用 CNN 的主要瓶颈还是在训练，用 CNN 做预测问题不大。目前 TensorFlow 已经支持在 iOS、Android 系统中运行，所以在手机上使用 CPU 进行人脸识别或图片分类已经非常方便了，并且响应速度也很快。

至此，AlexNet 的 TensorFlow 实现和运算时间评测就完成了。AlexNet 为卷积神经网

络和深度学习正名，以绝对优势拿下 ILSVRC 2012 冠军，引起了学术界的极大关注，为复兴神经网络做出了很大贡献。AlexNet 在 ILSVRC 数据集上可达到 16.4% 的错误率（读者可自行下载数据集测试，但注意 batch_size 可能要设为 1 才能复现论文中的结果），其中用到的许多网络结构和 Trick 给深度学习的发展带来了深刻的影响。当然，我们也不能忽视 ImageNet 数据集给深度学习带来的贡献。训练深度卷积神经网络，必须拥有一个像 ImageNet 这样超大的数据集才能避免过拟合，发挥深度学习的优势。可以说，传统机器学习模型适合学习一个小型数据集，但是对于大型数据集，我们需要有更大学习容量（Learning Capacity）的模型，即深度学习模型。

在 AlexNet 发表在 NIPS 时，Hinton 曾说“如果你没有参加过之前十几年的 NIPS，那没关系，因为直到今年神经网络才真正开始生效。”当时有很多与会的教授表示不能接受神经网络，他们认为深度学习是一个黑箱模型，不可解释，有超过 6000 万的参数，在 ILSVRC 上的成绩可能只是某种过拟合，对计算机视觉贡献不大。然而之前使用传统方法获得过 ILSVRC 比赛冠军的 NEC Labs 的模型参数量也不少。他们当时使用 sparse SIFT 加 Pyramid Pooling 提取特征，然后使用 SVM 进行分类，但是他们的 SVM 模型参数也有超过 1 亿 6000 万，远比 AlexNet 的参数多。因此说 CNN 参数多所以没有价值的观点是站不住脚的。深度学习的参数并不一定比传统机器学习模型多，尤其卷积层使用的参数量其实很少，但是其抽取特征的能力是非常强的，这也是 CNN 之所以有效的原因。

6.2 TensorFlow 实现 VGGNet

VGGNet 是牛津大学计算机视觉组（Visual Geometry Group）和 Google DeepMind 公司的研究员一起研发的深度卷积神经网络。VGGNet 探索了卷积神经网络的深度与其性能之间的关系，通过反复堆叠 3×3 的小型卷积核和 2×2 的最大池化层，VGGNet 成功地构筑了 16~19 层深的卷积神经网络。VGGNet 相比之前 state-of-the-art 的网络结构，错误率大幅下降，并取得了 ILSVRC 2014 比赛分类项目的第 2 名和定位项目的第 1 名。同时 VGGNet 的拓展性很强，迁移到其他图片数据上的泛化性非常好。VGGNet 的结构非常简洁，整个网络都使用了同样大小的卷积核尺寸（ 3×3 ）和最大池化尺寸（ 2×2 ）。到目前为止，VGGNet 依然经常被用来提取图像特征。VGGNet 训练后的模型参数在其官方网站上开源了，可用来在 domain specific 的图像分类任务上进行再训练（相当于提供了非常好的初始化权重），因此被用在了很多地方。

VGGNet 论文中全部使用了 3×3 的卷积核和 2×2 的池化核, 通过不断加深网络结构来提升性能。图 6-6 所示为 VGGNet 各级别的网络结构图, 图 6-7 所示为每一级别的参数量, 从 11 层的网络一直到 19 层的网络都有详尽的性能测试。虽然从 A 到 E 每一级网络逐渐变深, 但是网络的参数量并没有增长很多, 这是因为参数量主要都消耗在最后 3 个全连接层。前面的卷积部分虽然很深, 但是消耗的参数量不大, 不过训练比较耗时的部分依然是卷积, 因其计算量比较大。这其中的 D、E 也就是我们常说的 VGGNet-16 和 VGGNet-19。C 很有意思, 相比 B 多了几个 1×1 的卷积层, 1×1 卷积的意义主要在于线性变换, 而输入通道数和输出通道数不变, 没有发生降维。

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					

图 6-6 VGGNet 各级别网络结构图

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

图 6-7 VGGNet 各级别网络参数量 (单位为百万)

VGGNet 拥有 5 段卷积, 每一段内有 2~3 个卷积层, 同时每段尾部会连接一个最大池

化层用来缩小图片尺寸。每段内的卷积核数量一样，越靠后的段的卷积核数量越多：64 – 128 – 256 – 512 – 512。其中经常出现多个完全一样的 3×3 的卷积层堆叠在一起的情况，这其实是非常有用的设计。如图 6-8 所示，两个 3×3 的卷积层串联相当于 1 个 5×5 的卷积层，即一个像素会跟周围 5×5 的像素产生关联，可以说感受野大小为 5×5 。而 3 个 3×3 的卷积层串联的效果则相当于 1 个 7×7 的卷积层。除此之外，3 个串联的 3×3 的卷积层，拥有比 1 个 7×7 的卷积层更少的参数量，只有后者的 $\frac{3 \times 3 \times 3}{7 \times 7} = 55\%$ 。最重要的是，3 个 3×3 的卷积层拥有比 1 个 7×7 的卷积层更多的非线性变换（前者可以使用三次 ReLU 激活函数，而后者只有一次），使得 CNN 对特征的学习能力更强。

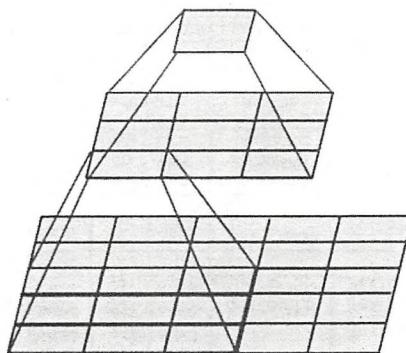


图 6-8 两个串联 3×3 的卷积层功能类似于一个 5×5 的卷积层

VGGNet 在训练时有一个小技巧，先训练级别 A 的简单网络，再复用 A 网络的权重来初始化后面的几个复杂模型，这样训练收敛的速度更快。在预测时，VGG 采用 Multi-Scale 的方法，将图像 scale 到一个尺寸 Q，并将图片输入卷积网络计算。然后在最后一个卷积层使用滑窗的方式进行分类预测，将不同窗口的分类结果平均，再将不同尺寸 Q 的结果平均得到最后结果，这样可提高图片数据的利用率并提升预测准确率。同时在训练中，VGGNet 还使用了 Multi-Scale 的方法做数据增强，将原始图像缩放到不同尺寸 S，然后再随机裁切 224×224 的图片，这样能增加很多数据量，对于防止模型过拟合有很不错的效果。实践中，作者令 S 在 [256, 512] 这个区间内取值，使用 Multi-Scale 获得多个版本的数据，并将多个版本的数据合在一起进行训练。图 6-9 所示为 VGGNet 使用 Multi-Scale 训练时得到的结果，可以看到 D 和 E 都可以达到 7.5% 的错误率。最终提交到 ILSVRC 2014 的版本是仅使用 Single-Scale 的 6 个不同等级的网络与 Multi-Scale 的 D 网络的融合，达到了 7.3% 的错误率。不过比赛结束后作者发现只融合 Multi-Scale 的 D 和 E 可以达到更好的效果，错误率达到 7.0%，再使用其他优化策略最终错误率可达到 6.8% 左右，非常接近同

年的冠军 Google Inception Net。同时，作者在对比各级网络时总结出了以下几个观点。

- (1) LRN 层作用不大。
- (2) 越深的网络效果越好。
- (3) 1×1 的卷积也是很有效的，但是没有 3×3 的卷积好，大一些的卷积核可以学习更大的空间特征。

ConvNet config. (Table 1)	smallest image side		top-1 val. error (%)	top-5 val. error (%)
	train (S)	test (Q)		
B	256	224,256,288	28.2	9.6
	256	224,256,288	27.7	9.2
C	384	352,384,416	27.8	9.2
	[256; 512]	256,384,512	26.3	8.2
D	256	224,256,288	26.6	8.6
	384	352,384,416	26.5	8.6
E	[256; 512]	256,384,512	24.8	7.5
	256	224,256,288	26.9	8.7
F	384	352,384,416	26.7	8.6
	[256; 512]	256,384,512	24.8	7.5

图 6-9 各级别 VGGNet 在使用 Multi-Scale 训练时的 top-5 错误率

VGGNet 训练时使用了 4 块 Geforce GTX Titan GPU 并行计算，速度比单块 GPU 快 3.75 倍，几乎没有太多性能损耗。但是，每个网络耗时 2~3 周才可以训练完。因此我们这里不直接使用 ImageNet 数据训练一个 VGGNet，而是采用跟 AlexNet 一样的方式：构造出 VGGNet 网络结构，并评测其 forward (inference) 耗时和 backward (training) 耗时。

下面就开始实现 VGGNet-16，也就是上面的版本 D，其他版本读者可以仿照本节的代码自行修改并实现，难度不大。首先，我们载入几个系统库和 TensorFlow。本节代码主要来自 tensorflow-vgg 的开源实现⁵²。

```
from datetime import datetime
import math
import time
import tensorflow as tf
```

VGGNet-16 包含很多层的卷积，因此我们先写一个函数 conv_op，用来创建卷积层并把本层的参数存入参数列表。先来看 conv_op 函数的输入，input_op 是输入的 tensor，name 是这一层的名称，kh 是 kernel height 即卷积核的高，kw 是 kernel width 即卷积核的宽，n_out 是卷积核数量即输出通道数，dh 是步长的高，dw 是步长的宽，p 是参数列表。下面使用 get_shape()[-1].value 获取输入 input_op 的通道数，比如输入图片的尺寸 $224 \times 224 \times 3$

中最后的那个 3。然后使用 `tf.name_scope(name)` 设置 scope。我们的 `kernel`(即卷积核参数) 使用 `tf.get_variable` 创建，其中 `shape` 就是 `[kh, kw, n_in, n_out]` 即 [卷积核的高, 卷积核的宽, 输入通道数, 输出通道数]，同时使用 `tf.contrib.layers.xavier_initializer_conv2d()` 做参数初始化。Xavier 初始化方法我们在第 4 章实现过，其原理也讲解过，在此不做赘述。

```
def conv_op(input_op, name, kh, kw, n_out, dh, dw, p):
    n_in = input_op.get_shape()[-1].value

    with tf.name_scope(name) as scope:
        kernel = tf.get_variable(scope+"w",
            shape=[kh, kw, n_in, n_out], dtype=tf.float32,
            initializer=tf.contrib.layers.xavier_initializer_conv2d())
```

接着使用 `tf.nn.conv2d` 对 `input_op` 进行卷积处理，卷积核即为 `kernel`，步长是 `dh×dw`，`padding` 模式设为 `SAME`。`biases` 使用 `tf.constant` 赋值为 0，再使用 `tf.Variable` 将其转成可训练的参数。我们使用 `tf.nn.bias_add` 将卷积结果 `conv` 与 `bias` 相加，再使用 `tf.nn.relu` 对其进行非线性处理得到 `activation`。最后将创建卷积层时用到的参数 `kernel` 和 `biases` 添加进参数列表 `p`，并将卷积层的输出 `activation` 作为函数结果返回。

```
conv = tf.nn.conv2d(input_op, kernel, (1, dh, dw, 1),
                    padding='SAME')
bias_init_val = tf.constant(0.0, shape=[n_out], dtype=tf.float32)
biases = tf.Variable(bias_init_val, trainable=True, name='b')
z = tf.nn.bias_add(conv, biases)
activation = tf.nn.relu(z, name=scope)
p += [kernel, biases]
return activation
```

下面定义全连接层的创建函数 `fc_op`。一样是先获取输入 `input_op` 的通道数，然后使用 `tf.get_variable` 创建全连接层的参数，只不过参数的维度只有两个，第一个维度为输入的通道数 `n_in`，第二个维度为输出的通道数 `n_out`。同样，参数初始化方法也使用 `xavier_initializer`。这里 `biases` 不再初始化为 0，而是赋予一个较小的值 0.1 以避免 dead neuron。然后使用 `tf.nn.relu_layer` 对输入变量 `input_op` 与 `kernel` 做矩阵乘法并加上 `biases`，再做 ReLU 非线性变换得到 `activation`。最后将这个全连接层用到参数 `kernel`、`biases` 添加

到参数列表 p，并将 activation 作为函数结果返回。

```
def fc_op(input_op, name, n_out, p):
    n_in = input_op.get_shape()[-1].value

    with tf.name_scope(name) as scope:
        kernel = tf.get_variable(scope+"w",
            shape=[n_in, n_out], dtype=tf.float32,
            initializer=tf.contrib.layers.xavier_initializer())
        biases = tf.Variable(tf.constant(0.1, shape=[n_out]),
            dtype= tf.float32, name='b')
        activation = tf.nn.relu_layer(input_op, kernel, biases, name= scope)
        p += [kernel, biases]
    return activation
```

再定义最大池化层的创建函数 mpool_op。这里直接使用 tf.nn.max_pool，输入即为 input_op，池化尺寸为 kh×kw，步长是 dh×dw，padding 模式设为 SAME。

```
def mpool_op(input_op, name, kh, kw, dh, dw):
    return tf.nn.max_pool(input_op,
        ksize=[1, kh, kw, 1],
        strides=[1, dh, dw, 1],
        padding='SAME',
        name=name)
```

完成了卷积层、全连接层和最大池化层的创建函数，接下来就开始创建 VGGNet-16 的网络结构。VGGNet-16 主要分为 6 个部分，前 5 段为卷积网络，最后一段是全连接网络。我们定义创建 VGGNet-16 网络结构的函数 inference_op，输入有 input_op 和 keep_prob，这里的 keep_prob 是控制 dropout 比率的一个 placeholder。第一步先初始化参数列表 p。然后创建第一段卷积网络，这一段正如图 6-6 中的网络结构，由两个卷积层和一个最大池化层构成。我们使用前面写好的函数 conv_op、mpool_op 来创建他们。这两个卷积层的卷积核的大小都是 3×3，同时卷积核数量（输出通道数）均为 64，步长为 1×1，全像素扫描。第一个卷积层的输入 input_op 的尺寸为 224×224×3，输出尺寸为 224×224×64；而第二个卷积层的输入输出尺寸均为 224×224×64。卷积层后的最大池化层则是一个标准的 2×2 的最大池化，将输出结果尺寸变为了 112×112×64。

```

def inference_op(input_op, keep_prob):
    p = []

    conv1_1 = conv_op(input_op, name="conv1_1", kh=3, kw=3, n_out=64, dh=1,
                      dw=1, p=p)
    conv1_2 = conv_op(conv1_1, name="conv1_2", kh=3, kw=3, n_out=64, dh=1,
                      dw=1, p=p)
    pool1 = mpool_op(conv1_2, name="pool1", kh=2, kw=2, dw=2, dh=2)

```

第二段卷积网络和第一段非常类似，同样是两个卷积层加一个最大池化层，两个卷积层的卷积核尺寸也是 3×3 ，但是输出通道数变为128，是以前的两倍。最大池化层则和前面保持一致，因此这一段卷积网络的输出尺寸变为 $56\times 56\times 128$ 。

```

conv2_1 = conv_op(pool1, name="conv2_1", kh=3, kw=3, n_out=128, dh=1,
                  dw=1, p=p)
conv2_2 = conv_op(conv2_1, name="conv2_2", kh=3, kw=3, n_out=128, dh=1,
                  dw=1, p=p)
pool2 = mpool_op(conv2_2, name="pool2", kh=2, kw=2, dh=2, dw=2)

```

接下来是第三段卷积网络，这里有3个卷积层和1个最大池化层。3个卷积层的卷积核大小依然是 3×3 ，但是输出通道数增长为256，而最大池化层保持不变，因此这一段卷积网络的输出尺寸是 $28\times 28\times 256$ 。

```

conv3_1 = conv_op(pool2, name="conv3_1", kh=3, kw=3, n_out=256, dh=1,
                  dw=1, p=p)
conv3_2 = conv_op(conv3_1, name="conv3_2", kh=3, kw=3, n_out=256, dh=1,
                  dw=1, p=p)
conv3_3 = conv_op(conv3_2, name="conv3_3", kh=3, kw=3, n_out=256, dh=1,
                  dw=1, p=p)
pool3 = mpool_op(conv3_3, name="pool3", kh=2, kw=2, dh=2, dw=2)

```

第四段卷积网络也是3个卷积层加1个最大池化层。读者可能已经发现规律了，到目前为止，VGGNet-16的每一段卷积网络都会将图像的边长缩小一半，但是将卷积输出通道数翻倍。这样图像面积缩小到 $1/4$ ，输出通道数变为2倍，因此输出tensor的总尺寸每次缩小一半。这一层就是将卷积输出通道数增加到512，但是通过最大池化将图片缩小为

14×14。

```
conv4_1 = conv_op(pool3, name="conv4_1", kh=3, kw=3, n_out=512, dh=1,
                  dw=1, p=p)
conv4_2 = conv_op(conv4_1, name="conv4_2", kh=3, kw=3, n_out=512, dh=1,
                  dw=1, p=p)
conv4_3 = conv_op(conv4_2, name="conv4_3", kh=3, kw=3, n_out=512, dh=1,
                  dw=1, p=p)
pool4 = mpool_op(conv4_3, name="pool4", kh=2, kw=2, dh=2, dw=2)
```

最后一段卷积网络有所变化，这里卷积输出的通道数不再增加，继续维持在 512。最后一段卷积网络同样是 3 个卷积层加一个最大池化层，卷积核尺寸为 3×3，步长为 1×1，池化层尺寸为 2×2，步长为 2×2。因此到这里输出的尺寸变为 7×7×512。

```
conv5_1 = conv_op(pool4, name="conv5_1", kh=3, kw=3, n_out=512, dh=1,
                  dw=1, p=p)
conv5_2 = conv_op(conv5_1, name="conv5_2", kh=3, kw=3, n_out=512, dh=1,
                  dw=1, p=p)
conv5_3 = conv_op(conv5_2, name="conv5_3", kh=3, kw=3, n_out=512, dh=1,
                  dw=1, p=p)
pool5 = mpool_op(conv5_3, name="pool5", kh=2, kw=2, dw=2, dh=2)
```

我们将第 5 段卷积网络的输出结果进行扁平化，使用 tf.reshape 函数将每个样本化为长度为 7×7×512=25088 的一维向量。

```
shp = pool5.get_shape()
flattened_shape = shp[1].value * shp[2].value * shp[3].value
resh1 = tf.reshape(pool5, [-1, flattened_shape], name="resh1")
```

然后连接一个隐含节点数为 4096 的全连接层，激活函数为 ReLU。然后连接一个 Dropout 层，在训练时节点保留率为 0.5，预测时为 1.0。

```
fc6 = fc_op(resh1, name="fc6", n_out=4096, p=p)
fc6_drop = tf.nn.dropout(fc6, keep_prob, name="fc6_drop")
```

接下来是一个和前面一样的全连接层，之后同样连接一个 Dropout 层。

```
fc7 = fc_op(fc6_drop, name="fc7", n_out=4096, p=p)
```

```
fc7_drop = tf.nn.dropout(fc7, keep_prob, name="fc7_drop")
```

最后连接一个有 1000 个输出节点的全连接层，并使用 Softmax 进行处理得到分类输出概率。这里使用 `tf.argmax` 求输出概率最大的类别。最后将 `fc8`、`softmax`、`predictions` 和参数列表 `p` 一起返回。到此为止，VGGNet-16 的网络结构就全部构建完成了。

```
fc8 = fc_op(fc7_drop, name="fc8", n_out=1000, p=p)
softmax = tf.nn.softmax(fc8)
predictions = tf.argmax(softmax, 1)
return predictions, softmax, fc8, p
```

我们的评测函数 `time_tensorflow_run()` 和前面 AlexNet 中的非常相似，只有一点区别：我们在 `session.run()` 方法中引入了 `feed_dict`，方便后面传入 `keep_prob` 来控制 Dropout 层的保留比率。

```
def time_tensorflow_run(session, target, feed, info_string):
    num_steps_burn_in = 10
    total_duration = 0.0
    total_duration_squared = 0.0
    for i in range(num_batches + num_steps_burn_in):
        start_time = time.time()
        _ = session.run(target, feed_dict=feed)
        duration = time.time() - start_time
        if i >= num_steps_burn_in:
            if not i % 10:
                print ('%s: step %d, duration = %.3f' %
                       (datetime.now(), i - num_steps_burn_in, duration))
            total_duration += duration
            total_duration_squared += duration * duration
    mn = total_duration / num_batches
    vr = total_duration_squared / num_batches - mn * mn
    sd = math.sqrt(vr)
    print ('%s: %s across %d steps, %.3f +/- %.3f sec / batch' %
           (datetime.now(), info_string, num_batches, mn, sd))
```

下面定义评测的主函数 `run_benchmark`，我们的目标依然是仅评测 forward 和 backward

的运算性能，并不进行实质的训练和预测。首先是生成尺寸为 224×224 的随机图片，方法与 AlexNet 中一样，通过 `tf.random_normal` 函数生成标准差为 0.1 的正态分布的随机数。

```
def run_benchmark():
    with tf.Graph().as_default():
        image_size = 224
        images = tf.Variable(tf.random_normal([batch_size,
                                              image_size,
                                              image_size, 3],
                                              dtype=tf.float32,
                                              stddev=1e-1))
```

接下来，创建 `keep_prob` 的 placeholder，并调用 `inference_op` 函数构建 VGGNet-16 的网络结构，获得 `predictions`、`softmax`、`fc8` 和参数列表 `p`。

```
keep_prob = tf.placeholder(tf.float32)
predictions, softmax, fc8, p = inference_op(images, keep_prob)
```

然后创建 Session 并初始化全局参数。

```
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)
```

我们通过将 `keep_prob` 设为 1.0 来执行预测，并使用 `time_tensorflow_run` 评测 forward 运算时间。再计算 VGGNet-16 最后的全连接层的输出 `fc8` 的 l2 loss，并使用 `tf.gradients` 求相对于这个 loss 的所有模型参数的梯度。最后使用 `time_tensorflow_run` 评测 backward 运算时间，这里 `target` 为求解梯度的操作 `grad`，`keep_prob` 为 0.5。

```
time_tensorflow_run(sess, predictions, {keep_prob:1.0}, "Forward")
objective = tf.nn.l2_loss(fc8)
grad = tf.gradients(objective, p)
time_tensorflow_run(sess, grad, {keep_prob:0.5}, "Forward-backward")
```

我们设置 `batch_size` 为 32，因为 VGGNet-16 的模型体积比较大，如果使用较大的 `batch_size`，GPU 显存会不够用。最后执行评测的主函数 `run_benchmark()`，测试 VGGNet-16 在 TensorFlow 上的 forward 和 backward 耗时。

```
batch_size=32  
num_batches=100  
run_benchmark()
```

forward 计算时平均每个 batch 的耗时为 0.152s, 相比于同样 batch size 的 AlexNet 的 0.026s(如果无 LRN 则是 0.007s)慢 6 倍。这说明 VGGNet-16 的计算复杂度相比 AlexNet 确实高了很多, 不过同样也带来了很大的准确率提升。

```
2016-12-01 17:13:31.761549: step 0, duration = 0.151  
2016-12-01 17:13:33.281319: step 10, duration = 0.151  
2016-12-01 17:13:34.789695: step 20, duration = 0.151  
2016-12-01 17:13:36.311346: step 30, duration = 0.158  
2016-12-01 17:13:37.824452: step 40, duration = 0.153  
2016-12-01 17:13:39.341594: step 50, duration = 0.152  
2016-12-01 17:13:40.859964: step 60, duration = 0.152  
2016-12-01 17:13:42.379664: step 70, duration = 0.153  
2016-12-01 17:13:43.900802: step 80, duration = 0.153  
2016-12-01 17:13:45.424529: step 90, duration = 0.151  
2016-12-01 17:13:46.795223: Forward across 100 steps, 0.152 +/- 0.002 sec /  
batch
```

而 backward 求解梯度时, 每个 batch 的平均耗时达到 0.617s, 相比于 AlexNet 的 0.078s 也高了很多。

```
2016-12-01 17:13:57.078991: step 0, duration = 0.613  
2016-12-01 17:14:03.241287: step 10, duration = 0.621  
2016-12-01 17:14:09.398178: step 20, duration = 0.616  
2016-12-01 17:14:15.555161: step 30, duration = 0.617  
2016-12-01 17:14:21.713196: step 40, duration = 0.614  
2016-12-01 17:14:27.879734: step 50, duration = 0.614  
2016-12-01 17:14:34.044447: step 60, duration = 0.614  
2016-12-01 17:14:40.204176: step 70, duration = 0.619  
2016-12-01 17:14:46.373392: step 80, duration = 0.615  
2016-12-01 17:14:52.550798: step 90, duration = 0.620  
2016-12-01 17:14:58.123548: Forward-backward across 100 steps, 0.617 +/- 0.0
```

02 sec / batch

至此 VGGNet-16 的实现和评测就完成了。VGG 系列的卷积神经网络在 ILSVRC 2014 比赛中最终达到了 7.3% 的错误率，相比 AlexNet 进步非常大，读者可以使用 ImageNet 数据集复现其结果。VGGNet 的模型参数虽然比 AlexNet 多，但反而只需要较少的迭代次数就可以收敛，主要原因是更深的网络和更小的卷积核带来的隐式的正则化效果。VGGNet 凭借其相对不算很高的复杂度和优秀的分类性能，成为了一代经典的卷积神经网络，直到现在依然被应用在很多地方。

6.3 TensorFlow 实现 Google Inception Net

Google Inception Net 首次出现在 ILSVRC 2014 的比赛中（和 VGGNet 同年），就以较大优势取得了第一名。那届比赛中的 Inception Net 通常被称为 Inception V1，它最大的特点是控制了计算量和参数量的同时，获得了非常好的分类性能——top-5 错误率 6.67%，只有 AlexNet 的一半不到。Inception V1 有 22 层深，比 AlexNet 的 8 层或者 VGGNet 的 19 层还要更深。但其计算量只有 15 亿次浮点运算，同时只有 500 万的参数量，仅为 AlexNet 参数量（6000 万）的 1/12，却可以达到远胜于 AlexNet 的准确率，可以说是非常优秀并且非常实用的模型。Inception V1 降低参数量的目的有两点，第一，参数越多模型越庞大，需要供模型学习的数据量就越大，而目前高质量的数据非常昂贵；第二，参数越多，耗费的计算资源也会更大。Inception V1 参数少但效果好的原因除了模型层数更深、表达能力更强外，还有两点：一是去除了最后的全连接层，用全局平均池化层（即将图片尺寸变为 1×1 ）来取代它。全连接层几乎占据了 AlexNet 或 VGGNet 中 90% 的参数量，而且会引起过拟合，去除全连接层后模型训练更快并且减轻了过拟合。用全局平均池化层取代全连接层的做法借鉴了 *Network In Network*（以下简称 NIN）论文。二是 Inception V1 中精心设计的 Inception Module 提高了参数的利用效率，其结构如图 6-10 所示。这一部分也借鉴了 NIN 的思想，形象的解释就是 Inception Module 本身如同大网络中的一个小网络，其结构可以反复堆叠在一起形成大网络。不过 Inception V1 比 NIN 更进一步的是增加了分支网络，NIN 则主要是级联的卷积层和 MLPConv 层。一般来说卷积层要提升表达能力，主要依靠增加输出通道数，但副作用是计算量增大和过拟合。每一个输出通道对应一个滤波器，同一个滤波器共享参数，只能提取一类特征，因此一个输出通道只能做一种特征处理。而 NIN 中的 MLPConv 则拥有更强大的能力，允许在输出通道之间组合信息，因此效果明显。可以说，MLPConv 基本等效于普通卷积层后再连接 1×1 的卷积和 ReLU 激活

函数。

我们再来看 Inception Module 的基本结构，其中有 4 个分支：第一个分支对输入进行 1×1 的卷积，这其实也是 NIN 中提出的一个重要结构。 1×1 的卷积是一个非常优秀的结构，它可以跨通道组织信息，提高网络的表达能力，同时可以对输出通道升维和降维。可以看到 Inception Module 的 4 个分支都用到了 1×1 卷积，来进行低成本（计算量比 3×3 小很多）的跨通道的特征变换。第二个分支先使用了 1×1 卷积，然后连接 3×3 卷积，相当于进行了两次特征变换。第三个分支类似，先是 1×1 的卷积，然后连接 5×5 卷积。最后一个分支则是 3×3 最大池化后直接使用 1×1 卷积。我们可以发现，有的分支只使用 1×1 卷积，有的分支使用了其他尺寸的卷积时也会再使用 1×1 卷积，这是因为 1×1 卷积的性价比很高，用很小的计算量就能增加一层特征变换和非线性化。Inception Module 的 4 个分支在最后通过一个聚合操作合并（在输出通道数这个维度上聚合）。Inception Module 中包含了 3 种不同尺寸的卷积和 1 个最大池化，增加了网络对不同尺度的适应性，这一部分和 Multi-Scale 的思想类似。早期计算机视觉的研究中，受灵长类神经视觉系统的启发，Serre 使用不同尺寸的 Gabor 滤波器处理不同尺寸的图片，Inception V1 借鉴了这种思想。Inception V1 的论文中指出，Inception Module 可以让网络的深度和宽度高效率地扩充，提升准确率且不致于过拟合。

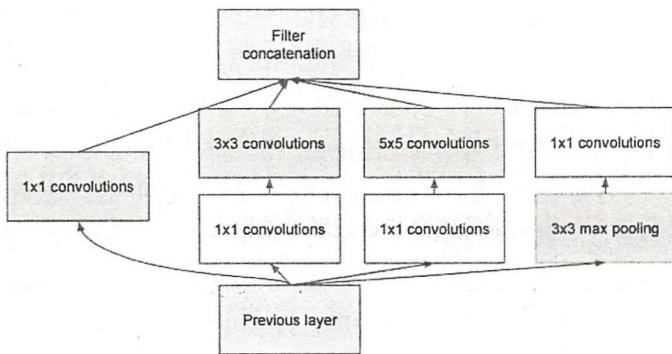


图 6-10 Inception Module 结构图

人脑神经元的连接是稀疏的，因此研究者认为大型神经网络的合理的连接方式应该也是稀疏的。稀疏结构是非常适合神经网络的一种结构，尤其是对非常大型、非常深的神经网络，可以减轻过拟合并降低计算量，例如卷积神经网络就是稀疏的连接。Inception Net 的主要目标就是找到最优的稀疏结构单元（即 Inception Module），论文中提到其稀疏结构基于 Hebbian 原理，这里简单解释一下 Hebbian 原理：神经反射活动的持续与重复会导

致神经元连接稳定性的持久提升，当两个神经元细胞 A 和 B 距离很近，并且 A 参与了对 B 重复、持续的兴奋，那么某些代谢变化会导致 A 将作为能使 B 兴奋的细胞。总结一下即“一起发射的神经元会连在一起”(Cells that fire together, wire together)，学习过程中的刺激会使神经元间的突触强度增加。受 Hebbian 原理启发，另一篇文章 *Provable Bounds for Learning Some Deep Representations* 提出，如果数据集的概率分布可以被一个很大很稀疏的神经网络所表达，那么构筑这个网络的最佳方法是逐层构筑网络：将上一层高度相关 (correlated) 的节点聚类，并将聚类出来的每一个小簇 (cluster) 连接到一起，如图 6-11 所示。这个相关性高的节点应该被连接在一起的结论，即是从神经网络的角度对 Hebbian 原理有效性的证明。

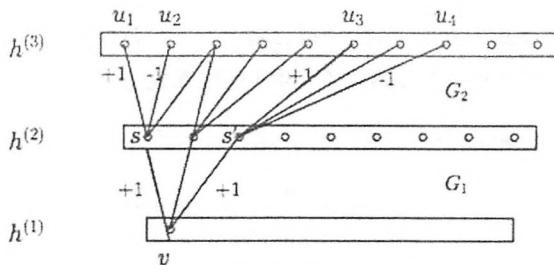


图 6-11 将高度相关的节点连接在一起，形成稀疏网络

因此一个“好”的稀疏结构，应该是符合 Hebbian 原理的，我们应该把相关性高的一簇神经元节点连接在一起。在普通的数据集中，这可能需要对神经元节点聚类，但是在图片数据中，天然的就是临近区域的数据相关性高，因此相邻的像素点被卷积操作连接在一起。而我们可能有多个卷积核，在同一空间位置但在不同通道的卷积核的输出结果相关性极高。因此，一个 1×1 的卷积就可以很自然地把这些相关性很高的、在同一个空间位置但是不同通道的特征连接在一起，这就是为什么 1×1 卷积这么频繁地被应用到 Inception Net 中的原因。 1×1 卷积所连接的节点的相关性是最高的，而稍微大一点尺寸的卷积，比如 3×3 、 5×5 的卷积所连接的节点相关性也很高，因此也可以适当地使用一些大尺寸的卷积，增加多样性 (diversity)。最后 Inception Module 通过 4 个分支中不同尺寸的 1×1 、 3×3 、 5×5 等小型卷积将相关性很高的节点连接在一起，就完成了其设计初衷，构建出了很高效的符合 Hebbian 原理的稀疏结构。

在 Inception Module 中，通常 1×1 卷积的比例（输出通道数占比）最高， 3×3 卷积和 5×5 卷积稍低。而在整个网络中，会有多个堆叠的 Inception Module，我们希望靠后的 Inception Module 可以捕捉更高阶的抽象特征，因此靠后的 Inception Module 的卷积的空

间集中度应该逐渐降低，这样可以捕获更大面积的特征。因此，越靠后的 Inception Module 中， 3×3 和 5×5 这两个大面积的卷积核的占比（输出通道数）应该更多。

Inception Net 有 22 层深，除了最后一层的输出，其中间节点的分类效果也很好。因此在 Inception Net 中，还使用到了辅助分类节点（auxiliary classifiers），即将中间某一层的输出用作分类，并按一个较小的权重（0.3）加到最终分类结果中。这样相当于做了模型融合，同时给网络增加了反向传播的梯度信号，也提供了额外的正则化，对于整个 Inception Net 的训练很有裨益。

当年的 Inception V1 还是跑在 TensorFlow 的前辈 DistBelief 上的，并且只运行在 CPU 上。当时使用了异步的 SGD 训练，学习速率每迭代 8 个 epoch 降低 4%。同时，Inception V1 也使用了 Multi-Scale、Multi-Crop 等数据增强方法，并在不同的采样数据上训练了 7 个模型进行融合，得到了最后的 ILSVRC 2014 的比赛成绩——top-5 错误率 6.67%。

同时，Google Inception Net 还是一个大家族，包括：

- 2014 年 9 月的论文 *Going Deeper with Convolutions* 提出的 Inception V1 (top-5 错误率 6.67%)。
- 2015 年 2 月的论文 *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate* 提出的 Inception V2 (top-5 错误率 4.8%)。
- 2015 年 12 月的论文 *Rethinking the Inception Architecture for Computer Vision* 提出的 Inception V3 (top-5 错误率 3.5%)。
- 2016 年 2 月的论文 *Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning* 提出的 Inception V4 (top-5 错误率 3.08%)。

Inception V2 学习了 VGGNet，用两个 3×3 的卷积代替 5×5 的大卷积（用以降低参数量并减轻过拟合），还提出了著名的 Batch Normalization（以下简称 BN）方法。BN 是一个非常有效的正则化方法，可以让大型卷积网络的训练速度加快很多倍，同时收敛后的分类准确率也可以得到大幅提高。BN 在用于神经网络某层时，会对每一个 mini-batch 数据的内部进行标准化（normalization）处理，使输出规范化到 $N(0,1)$ 的正态分布，减少了 Internal Covariate Shift（内部神经元分布的改变）。BN 的论文指出，传统的深度神经网络在训练时，每一层的输入的分布都在变化，导致训练变得困难，我们只能使用一个很小的学习速率解决这个问题。而对每一层使用 BN 之后，我们就可以有效地解决这个问题，学习速率可以增大很多倍，达到之前的准确率所需要的迭代次数只有 1/14，训练时间大

大缩短。而达到之前的准确率后，可以继续训练，并最终取得远超于 Inception V1 模型的性能——top-5 错误率 4.8%，已经优于人眼水平。因为 BN 某种意义上还起到了正则化的作用，所以可以减少或者取消 Dropout，简化网络结构。

当然，只是单纯地使用 BN 获得的增益还不明显，还需要一些相应的调整：增大学习速率并加快学习衰减速度以适用 BN 规范化后的数据；去除 Dropout 并减轻 L2 正则（因 BN 已起到正则化的作用）；去除 LRN；更彻底地对训练样本进行 shuffle；减少数据增强过程中对数据的光学畸变（因为 BN 训练更快，每个样本被训练的次数更少，因此更真实的样本对训练更有帮助）。在使用了这些措施后，Inception V2 在训练达到 Inception V1 的准确率时快了 14 倍，并且模型在收敛时的准确率上限更高。

而 Inception V3 网络则主要有两方面的改造：一是引入了 Factorization into small convolutions 的思想，将一个较大的二维卷积拆成两个较小的一维卷积，比如将 7×7 卷积拆成 1×7 卷积和 7×1 卷积，或者将 3×3 卷积拆成 1×3 卷积和 3×1 卷积，如图 6-12 所示。一方面节约了大量参数，加速运算并减轻了过拟合（比将 7×7 卷积拆成 1×7 卷积和 7×1 卷积，比拆成 3 个 3×3 卷积更节约参数），同时增加了一层非线性扩展模型表达能力。论文中指出，这种非对称的卷积结构拆分，其结果比对称地拆为几个相同的小卷积核效果更明显，可以处理更多、更丰富的空间特征，增加特征多样性。

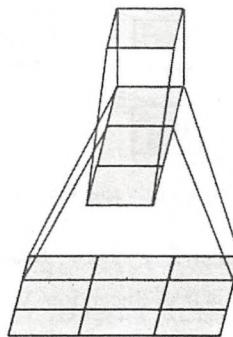


图 6-12 将一个 3×3 卷积拆成 1×3 卷积和 3×1 卷积

另一方面，Inception V3 优化了 Inception Module 的结构，现在 Inception Module 有 35×35 、 17×17 和 8×8 三种不同结构，如图 6-13 所示。这些 Inception Module 只在网络的后部出现，前部还是普通的卷积层。并且 Inception V3 除了在 Inception Module 中使用分支，还在分支中使用了分支（ 8×8 的结构中），可以说是 Network In Network In Network。

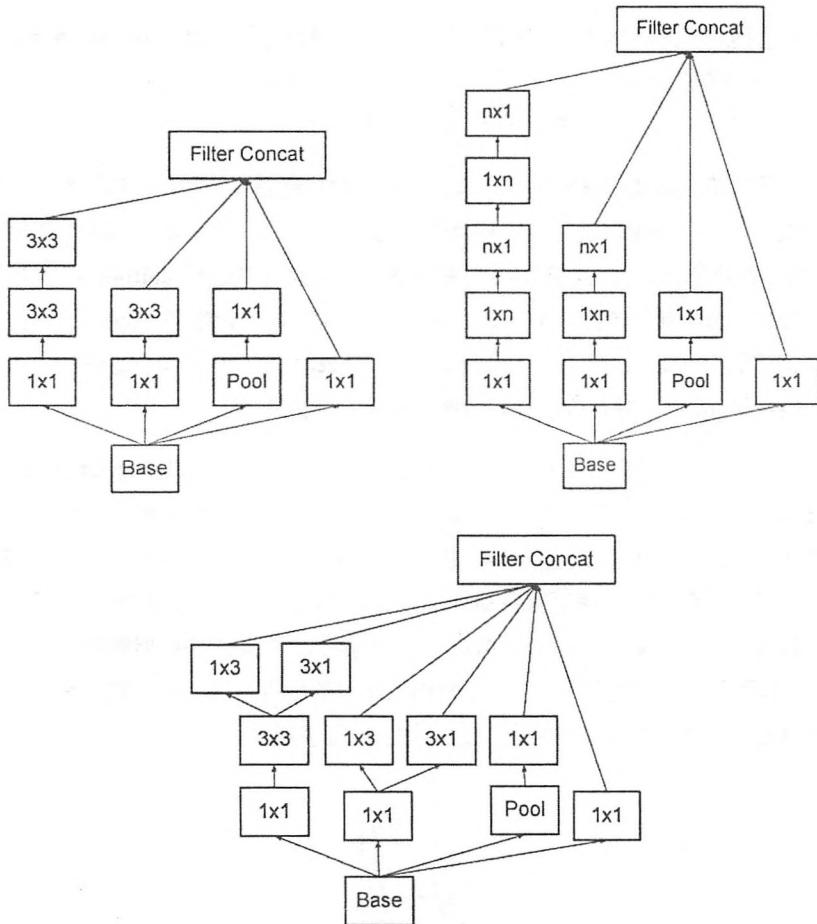


图 6-13 Inception V3 中三种结构的 Inception Module

而 Inception V4 相比 V3 主要是结合了微软的 ResNet，而 ResNet 将在 6.4 节单独讲解，这里不多做赘述。因此本节将实现的是 Inception V3，其整个网络结构如表 6-1 所示。由于 Google Inception Net V3 相对比较复杂，所以这里使用 `tf.contrib.slim` 辅助设计这个网络。`contrib.slim` 中的一些功能和组件可以大大减少设计 Inception Net 的代码量，我们只需要少量代码即可构建好有 42 层深的 Inception V3。

表 6-1 Inception V3 网络结构

类 型	kernel 尺寸/步长（或注释）	输入尺寸
卷积	$3 \times 3 / 2$	$299 \times 299 \times 3$

续表

类 型	kernel 尺寸/步长(或注释)	输入尺寸
卷积	3×3 / 1	149×149×32
卷积	3×3 / 1	147×147×32
池化	3×3 / 2	147×147×64
卷积	3×3 / 1	73×73×64
卷积	3×3 / 2	71×71×80
卷积	3×3 / 1	35×35×192
Inception 模块组	3 个 Inception Module	35×35×288
Inception 模块组	5 个 Inception Module	17×17×768
Inception 模块组	3 个 Inception Module	8×8×1280
池化	8×8	8×8×2048
线性	logits	1×1×2048
Softmax	分类输出	1×1×1000

首先定义一个简单的函数 `trunc_normal`, 产生截断的正态分布。本节代码主要来自 TensorFlow 的开源实现⁵³。

```
import tensorflow as tf
slim = tf.contrib.slim
trunc_normal = lambda stddev: tf.truncated_normal_initializer(0.0, stddev)
```

下面定义函数 `inception_v3_arg_scope`, 用来生成网络中经常用到的函数的默认参数, 比如卷积的激活函数、权重初始化方式、标准化器等。设置 L2 正则的 `weight_decay` 默认值为 0.00004, 标准差 `stddev` 默认值为 0.1, 参数 `batch_norm_var_collection` 默认值为 `moving_vars`。接下来, 定义 batch normalization 的参数字典, 定义其衰减系数 `decay` 为 0.9997, `epsilon` 为 0.001, `updates_collections` 为 `tf.GraphKeys.UPDATE_OPS`, 然后字典 `variables_collections` 中 `beta` 和 `gamma` 均设置为 `None`, `moving_mean` 和 `moving_variance` 均设置为前面的 `batch_norm_var_collection`。

接下来使用 `slim.arg_scope`, 这是一个非常有用的工具, 它可以给函数的参数自动赋予某些默认值。例如, 这句 `with slim.arg_scope([slim.conv2d, slim.fully_connected], weights_regularizer=slim.l2_regularizer(weight_decay))`, 会对 `[slim.conv2d, slim.fully_connected]` 这两个函数的参数自动赋值, 将参数 `weights_regularizer` 的值默认设为 `slim.l2_regularizer(weight_decay)`。使用了 `slim.arg_scope` 后就不需要每次都重复设置参数了, 只需要在有修改