

时设置。接下来，嵌套一个 `slim.arg_scope`，对卷积层生成函数 `slim.conv2d` 的几个参数赋予默认值，其权重初始化器 `weights_initializer` 设置为 `trunc_normal(stddev)`，激活函数设置为 `ReLU`，标准化器设置为 `slim.batch_norm`，标准化器的参数设置为前面定义的 `batch_norm_params`。最后返回定义好的 `scope`。

因为事先定义好了 `slim.conv2d` 中的各种默认参数，包括激活函数和标准化器，因此后面定义一个卷积层将会变得非常方便。我们可以用一行代码定义一个卷积层，整体代码会变得非常简洁美观，同时设计网络的工作量也会大大减轻。

```
def inception_v3_arg_scope(weight_decay=0.00004,
                           stddev=0.1,
                           batch_norm_var_collection='moving_vars'):

    batch_norm_params = {
        'decay': 0.9997,
        'epsilon': 0.001,
        'updates_collections': tf.GraphKeys.UPDATE_OPS,
        'variables_collections': {
            'beta': None,
            'gamma': None,
            'moving_mean': [batch_norm_var_collection],
            'moving_variance': [batch_norm_var_collection],
        }
    }

    with slim.arg_scope([slim.conv2d, slim.fully_connected],
                       weights_regularizer=slim.l2_regularizer(weight_decay)):
        with slim.arg_scope(
            [slim.conv2d],
            weights_initializer=tf.truncated_normal_initializer(stddev=stddev),
            activation_fn=tf.nn.relu,
            normalizer_fn=slim.batch_norm,
            normalizer_params=batch_norm_params) as sc:
            return sc
```

接下来我们就定义函数 `inception_v3_base`, 它可以生成 Inception V3 网络的卷积部分, 参数 `inputs` 为输入的图片数据的 `tensor`, `scope` 为包含了函数默认参数的环境。我们定义一个字典表 `end_points`, 用来保存某些关键节点供之后使用。接着再使用 `slim.arg_scope`, 对 `slim.conv2d`、`slim.max_pool2d` 和 `slim_avg_pool2d` 这三个函数的参数设置默认值, 将 `stride` 设为 1, `padding` 设为 `VALID`。下面正式开始定义 Inception V3 的网络结构, 首先是前面的非 Inception Module 的卷积层。这里直接使用 `slim.conv2d` 创建卷积层, `slim.conv2d` 的第 1 个参数为输入的 `tensor`, 第 2 个参数为输出的通道数, 第 3 个参数为卷积核尺寸, 第 4 个参数为步长 `stride`, 第 5 个参数为 `padding` 模式。我们的第一个卷积层的输出通道数为 32, 卷积核尺寸为 3×3 , 步长为 2, `padding` 模式则是默认的 `VALID`。后面的几个卷积层采用相同的形式, 按照论文中的定义, 逐层定义好网络结构。因为使用了 `slim` 及 `slim.arg_scope`, 我们一行代码就可以定义好一个卷积层, 相比之前 AlexNet 的实现中使用好几行代码定义一个卷积层, 或是 VGGNet 中专门写一个函数来定义卷积层, 都更加方便。

我们可以观察到, 在前面几个普通的非 Inception Module 的卷积层中, 主要使用了 3×3 的小卷积核, 这是充分借鉴了 VGGNet 的结构。同时, Inception V3 论文中也提出了 Factorization into small convolutions 思想, 利用两个 1 维卷积模拟大尺寸的 2 维卷积, 减少参数量同时增加非线性。前面几层卷积中还有一层 1×1 卷积, 这也是前面提到的 Inception Module 中经常使用的结构之一, 可低成本的跨通道的对特征进行组合。另外可以看到, 除了第一个卷积层步长为 2, 其余的卷积层步长均为 1, 而池化层则是尺寸为 3×3 、步长为 2 的重叠最大池化, 这是 AlexNet 中使用过的结构。网络的输入数据尺寸为 $299 \times 299 \times 3$, 在经历 3 个步长为 2 的层之后, 尺寸最后缩小为 $35 \times 35 \times 192$, 空间尺寸大大降低, 但是输出通道增加了很多。这部分代码中一共有 5 个卷积层, 2 个池化层, 实现了对输入图片数据的尺寸压缩, 并对图片特征进行了抽象。

```
def inception_v3_base(inputs, scope=None):
    end_points = {}
    with tf.variable_scope(scope, 'InceptionV3', [inputs]):
        with slim.arg_scope([slim.conv2d, slim.max_pool2d, slim.avg_pool2d],
                            stride=1, padding='VALID'):
            net = slim.conv2d(inputs, 32, [3, 3], stride=2, scope='Conv2d_1a_3x3')
            net = slim.conv2d(net, 32, [3, 3], scope='Conv2d_2a_3x3')
```

```

net = slim.conv2d(net, 64, [3, 3], padding='SAME',
                  scope='Conv2d_2b_3x3')
net = slim.max_pool2d(net, [3, 3], stride=2, scope='MaxPool_3a_3x3')
net = slim.conv2d(net, 80, [1, 1], scope='Conv2d_3b_1x1')
net = slim.conv2d(net, 192, [3, 3], scope='Conv2d_4a_3x3')
net = slim.max_pool2d(net, [3, 3], stride=2, scope='MaxPool_5a_3x3')

```

接下来就将是三个连续的 Inception 模块组，这三个 Inception 模块组中各自分别有多少个 Inception Module，这部分的网络结构即是 Inception V3 的精华所在。每个 Inception 模块组内部的几个 Inception Module 结构非常类似，但存在一些细节不同。

第 1 个 Inception 模块组包含了 3 个结构类似的 Inception Module，它们的结构和图 6-13 中第一幅图非常相似。其中第 1 个 Inception Module 的名称为 Mixed_5b。我们先使用 `slim.arg_scope` 设置所有 Inception 模块组的默认参数，将所有卷积层、最大池化、平均池化层的步长设为 1，`padding` 模式设为 SAME。然后设置这个 Inception Module 的 `variable_scope` 名称为 Mixed_5b。这个 Inception Module 中有 4 个分支，从 Branch_0 到 Branch_3，第一个分支为有 64 输出通道的 1×1 卷积；第 2 个分支为有 48 输出通道的 1×1 卷积，连接有 64 输出通道的 5×5 卷积；第 3 个分支为有 64 输出通道的 1×1 卷积，再连续连接 2 个有 96 输出通道的 3×3 卷积；第 4 个分支为 3×3 的平均池化，连接有 32 输出通道的 1×1 卷积。最后，使用 `tf.concat` 将 4 个分支的输出合并在一起（在第 3 个维度合并，即输出通道上合并），生成这个 Inception Module 的最终输出。因为这里所有的步长均为 1，并且 `padding` 模式为 SAME，所以图片的尺寸并不会缩小，依然维持在 35×35。不过通道数增加了，4 个分支的输出通道数之和 $64+64+96+32=256$ ，即最终输出的 tensor 尺寸为 35×35×256。这里需注意，第 1 个 Inception 模块组中所有 Inception Module 输出的图片尺寸均为 35×35，但是后两个 Inception Module 的通道数会发生变化。

```

with slim.arg_scope([slim.conv2d, slim.max_pool2d, slim.avg_pool2d],
                    stride=1, padding='SAME'):
    with tf.variable_scope('Mixed_5b'):
        with tf.variable_scope('Branch_0'):
            branch_0 = slim.conv2d(net, 64, [1, 1], scope='Conv2d_0a_1x1')
        with tf.variable_scope('Branch_1'):
            branch_1 = slim.conv2d(net, 48, [1, 1], scope='Conv2d_0a_1x1')
            branch_1 = slim.conv2d(branch_1, 64, [5, 5],

```

```

        scope='Conv2d_0b_5x5')

with tf.variable_scope('Branch_2'):
    branch_2 = slim.conv2d(net, 64, [1, 1], scope='Conv2d_0a_1x1')
    branch_2 = slim.conv2d(branch_2, 96, [3, 3],
                          scope='Conv2d_0b_3x3')
    branch_2 = slim.conv2d(branch_2, 96, [3, 3],
                          scope='Conv2d_0c_3x3')

with tf.variable_scope('Branch_3'):
    branch_3 = slim.avg_pool2d(net, [3, 3], scope='AvgPool_0a_3x3')
    branch_3 = slim.conv2d(branch_3, 32, [1, 1],
                          scope='Conv2d_0b_1x1')

net = tf.concat([branch_0, branch_1, branch_2, branch_3], 3)

```

接下来是第 1 个 Inception 模块组的第 2 个 Inception Module——Mixed_5c，这里依然使用前面设置的默认参数：步长为 1，padding 模式为 SAME。这个 Inception Module 同样有 4 个分支，唯一不同的是第 4 个分支最后接的是 64 输出通道的 1×1 卷积，而此前是 32 输出通道。因此，我们输出 tensor 的最终尺寸为 $35 \times 35 \times 288$ ，输出通道数相比之前增加了 32。

```

with tf.variable_scope('Mixed_5c'):
    with tf.variable_scope('Branch_0'):
        branch_0 = slim.conv2d(net, 64, [1, 1], scope='Conv2d_0a_1x1')

    with tf.variable_scope('Branch_1'):
        branch_1 = slim.conv2d(net, 48, [1, 1], scope='Conv2d_0b_1x1')
        branch_1 = slim.conv2d(branch_1, 64, [5, 5],
                              scope='Conv_1_0c_5x5')

    with tf.variable_scope('Branch_2'):
        branch_2 = slim.conv2d(net, 64, [1, 1], scope='Conv2d_0a_1x1')
        branch_2 = slim.conv2d(branch_2, 96, [3, 3],
                              scope='Conv2d_0b_3x3')
        branch_2 = slim.conv2d(branch_2, 96, [3, 3],
                              scope='Conv2d_0c_3x3')

    with tf.variable_scope('Branch_3'):
        branch_3 = slim.avg_pool2d(net, [3, 3], scope='AvgPool_0a_3x3')

```

```

branch_3 = slim.conv2d(branch_3, 64, [1, 1],
                      scope='Conv2d_0b_1x1')
net = tf.concat([branch_0, branch_1, branch_2, branch_3], 3)

```

而第 1 个 Inception 模块组的第 3 个 Inception Module——Mixed_5d 和上一个 Inception Module 完全相同，4 个分支的结构、参数一模一样，输出 tensor 的尺寸也为 $35 \times 35 \times 288$ 。

```

with tf.variable_scope('Mixed_5d'):
    with tf.variable_scope('Branch_0'):
        branch_0 = slim.conv2d(net, 64, [1, 1], scope='Conv2d_0a_1x1')
    with tf.variable_scope('Branch_1'):
        branch_1 = slim.conv2d(net, 48, [1, 1], scope='Conv2d_0a_1x1')
        branch_1 = slim.conv2d(branch_1, 64, [5, 5],
                              scope='Conv2d_0b_5x5')
    with tf.variable_scope('Branch_2'):
        branch_2 = slim.conv2d(net, 64, [1, 1], scope='Conv2d_0a_1x1')
        branch_2 = slim.conv2d(branch_2, 96, [3, 3],
                              scope='Conv2d_0b_3x3')
        branch_2 = slim.conv2d(branch_2, 96, [3, 3],
                              scope='Conv2d_0c_3x3')
    with tf.variable_scope('Branch_3'):
        branch_3 = slim.avg_pool2d(net, [3, 3], scope='AvgPool_0a_3x3')
        branch_3 = slim.conv2d(branch_3, 64, [1, 1],
                              scope='Conv2d_0b_1x1')
net = tf.concat([branch_0, branch_1, branch_2, branch_3], 3)

```

第 2 个 Inception 模块组是一个非常大的模块组，包含了 5 个 Inception Module，其中第 2 个到第 5 个 Inception Module 的结构非常类似，它们的结构如图 6-13 中第二幅图所示。其中第 1 个 Inception Module 名称为 Mixed_6a，它包含 3 个分支。第 1 个分支是一个 384 输出通道的 3×3 卷积，这个分支的通道数一下就超过了之前的通道数之和。不过步长为 2，因此图片尺寸将会被压缩，且 padding 模式为 VALID，所以图片尺寸缩小为 17×17 ；第 2 个分支有三层，分别是一个 64 输出通道的 1×1 卷积和两个 96 输出通道的 3×3 卷积。这里需要注意，最后一层的步长为 2，padding 模式为 VALID，因此图片尺寸也被压缩，

本分支最终输出的 tensor 尺寸为 $17 \times 17 \times 96$ ；第 3 个分支是一个 3×3 最大池化层，步长同样为 2，padding 模式为 VALID，因此输出的 tensor 尺寸为 $17 \times 17 \times 256$ 。最后依然是使用 tf.concat 将三个分支在输出通道上合并，最后的输出尺寸为 $17 \times 17 \times (384+96+256)=17 \times 17 \times 768$ 。在第 2 个 Inception 模块组中，5 个 Inception Module 输出 tensor 的尺寸将全部定格为 $17 \times 17 \times 768$ ，即图片尺寸和输出通道数都没有发生变化。

```

with tf.variable_scope('Mixed_6a'):
    with tf.variable_scope('Branch_0'):
        branch_0 = slim.conv2d(net, 384, [3, 3], stride=2,
                              padding='VALID', scope='Conv2d_1a_1x1')
    with tf.variable_scope('Branch_1'):
        branch_1 = slim.conv2d(net, 64, [1, 1], scope='Conv2d_0a_1x1')
        branch_1 = slim.conv2d(branch_1, 96, [3, 3],
                              scope='Conv2d_0b_3x3')
        branch_1 = slim.conv2d(branch_1, 96, [3, 3], stride=2,
                              padding='VALID', scope='Conv2d_1a_1x1')
    with tf.variable_scope('Branch_2'):
        branch_2 = slim.max_pool2d(net, [3, 3], stride=2, padding='VALID',
                                   scope='MaxPool_1a_3x3')
    net = tf.concat([branch_0, branch_1, branch_2], 3)

```

接下来是第 2 个 Inception 模块组的第 2 个 Inception Module——Mixed_6b，它有 4 个分支。第 1 个分支是一个简单的 192 输出通道的 1×1 卷积；第 2 个分支由 3 个卷积层组成，第 1 层是 128 输出通道的 1×1 卷积，第 2 层是 128 通道数的 1×7 卷积，第 3 层是 192 输出通道数的 7×1 卷积。这里即是前面提到的 Factorization into small convolutions 思想，串联的 1×7 卷积和 7×1 卷积相当于合成了一个 7×7 卷积，不过参数量大大减少了（只有后者的 $2/7$ ）并减轻了过拟合，同时多了一个激活函数增强了非线性特征变换；第 3 个分支一下子拥有了 5 个卷积层，分别是 128 输出通道的 1×1 卷积，128 输出通道的 7×1 卷积，128 输出通道的 1×7 卷积，128 输出通道的 7×1 卷积和 192 输出通道的 1×7 卷积。这个分支可以算是利用 Factorization into small convolutions 的典范，反复地将 7×7 卷积进行拆分；最后，第 4 个分支是一个 3×3 的平均池化层，再连接 192 输出通道的 1×1 卷积。最后将 4 个分支合并，这一层输出 tensor 的尺寸即是 $17 \times 17 \times (192+192+192+192)=17 \times 17 \times 768$ 。

```

with tf.variable_scope('Mixed_6b'):
    with tf.variable_scope('Branch_0'):
        branch_0 = slim.conv2d(net, 192, [1, 1], scope='Conv2d_0a_1x1')
    with tf.variable_scope('Branch_1'):
        branch_1 = slim.conv2d(net, 128, [1, 1], scope='Conv2d_0a_1x1')
        branch_1 = slim.conv2d(branch_1, 128, [1, 7],
                               scope='Conv2d_0b_1x7')
        branch_1 = slim.conv2d(branch_1, 192, [7, 1],
                               scope='Conv2d_0c_7x1')
    with tf.variable_scope('Branch_2'):
        branch_2 = slim.conv2d(net, 128, [1, 1], scope='Conv2d_0a_1x1')
        branch_2 = slim.conv2d(branch_2, 128, [7, 1],
                               scope='Conv2d_0b_7x1')
        branch_2 = slim.conv2d(branch_2, 128, [1, 7],
                               scope='Conv2d_0c_1x7')
        branch_2 = slim.conv2d(branch_2, 128, [7, 1],
                               scope='Conv2d_0d_7x1')
        branch_2 = slim.conv2d(branch_2, 192, [1, 7],
                               scope='Conv2d_0e_1x7')
    with tf.variable_scope('Branch_3'):
        branch_3 = slim.avg_pool2d(net, [3, 3], scope='AvgPool_0a_3x3')
        branch_3 = slim.conv2d(branch_3, 192, [1, 1],
                               scope='Conv2d_0b_1x1')
net = tf.concat([branch_0, branch_1, branch_2, branch_3], 3)

```

然后是我们第 2 个 Inception 模块组的第 3 个 Inception Module——Mixed_6c。Mixed_6c 和前面一个 Inception Module 非常相似，只有一个地方不同，即第 2 个分支和第 3 个分支中前几个卷积层的输出通道数不同，从 128 变为了 160，但是这两个分支的最终输出通道数不变，都是 192。其他地方则完全一致。需要注意的是，我们的网络每经过一个 Inception Module，即使输出 tensor 尺寸不变，但是特征都相当于被重新精炼了一遍，其中丰富的卷积和非线性化对提升网络性能帮助很大。

```

with tf.variable_scope('Mixed_6c'):
    with tf.variable_scope('Branch_0'):

```

```

branch_0 = slim.conv2d(net, 192, [1, 1], scope='Conv2d_0a_1x1')
with tf.variable_scope('Branch_1'):
    branch_1 = slim.conv2d(net, 160, [1, 1], scope='Conv2d_0a_1x1')
    branch_1 = slim.conv2d(branch_1, 160, [1, 7],
                          scope='Conv2d_0b_1x7')
    branch_1 = slim.conv2d(branch_1, 192, [7, 1],
                          scope='Conv2d_0c_7x1')
with tf.variable_scope('Branch_2'):
    branch_2 = slim.conv2d(net, 160, [1, 1], scope='Conv2d_0a_1x1')
    branch_2 = slim.conv2d(branch_2, 160, [7, 1],
                          scope='Conv2d_0b_7x1')
    branch_2 = slim.conv2d(branch_2, 160, [1, 7],
                          scope='Conv2d_0c_1x7')
    branch_2 = slim.conv2d(branch_2, 160, [7, 1],
                          scope='Conv2d_0d_7x1')
    branch_2 = slim.conv2d(branch_2, 192, [1, 7],
                          scope='Conv2d_0e_1x7')
with tf.variable_scope('Branch_3'):
    branch_3 = slim.avg_pool2d(net, [3, 3], scope='AvgPool_0a_3x3')
    branch_3 = slim.conv2d(branch_3, 192, [1, 1],
                          scope='Conv2d_0b_1x1')
net = tf.concat([branch_0, branch_1, branch_2, branch_3], 3)

```

Mixed_6d 和前面的 Mixed_6c 完全一致，目的同样是通过 Inception Module 精心设计的结构增加卷积和非线性，提炼特征。

```

with tf.variable_scope('Mixed_6d'):
    with tf.variable_scope('Branch_0'):
        branch_0 = slim.conv2d(net, 192, [1, 1], scope='Conv2d_0a_1x1')
    with tf.variable_scope('Branch_1'):
        branch_1 = slim.conv2d(net, 160, [1, 1], scope='Conv2d_0a_1x1')
        branch_1 = slim.conv2d(branch_1, 160, [1, 7],
                              scope='Conv2d_0b_1x7')
        branch_1 = slim.conv2d(branch_1, 192, [7, 1],

```

```

        scope='Conv2d_0c_7x1')

with tf.variable_scope('Branch_2'):
    branch_2 = slim.conv2d(net, 160, [1, 1], scope='Conv2d_0a_1x1')
    branch_2 = slim.conv2d(branch_2, 160, [7, 1],
                          scope='Conv2d_0b_7x1')
    branch_2 = slim.conv2d(branch_2, 160, [1, 7],
                          scope='Conv2d_0c_1x7')
    branch_2 = slim.conv2d(branch_2, 160, [7, 1],
                          scope='Conv2d_0d_7x1')
    branch_2 = slim.conv2d(branch_2, 192, [1, 7],
                          scope='Conv2d_0e_1x7')

with tf.variable_scope('Branch_3'):
    branch_3 = slim.avg_pool2d(net, [3, 3], scope='AvgPool_0a_3x3')
    branch_3 = slim.conv2d(branch_3, 192, [1, 1],
                          scope='Conv2d_0b_1x1')

net = tf.concat([branch_0, branch_1, branch_2, branch_3], 3)

```

Mixed_6e 也和前面两个 Inception Module 完全一致。这是第 2 个 Inception 模块组的最后一个 Inception Module。我们将 Mixed_6e 存储于 end_points 中，作为 Auxiliary Classifier 辅助模型的分类。

```

with tf.variable_scope('Mixed_6e'):
    with tf.variable_scope('Branch_0'):
        branch_0 = slim.conv2d(net, 192, [1, 1], scope='Conv2d_0a_1x1')
    with tf.variable_scope('Branch_1'):
        branch_1 = slim.conv2d(net, 192, [1, 1], scope='Conv2d_0a_1x1')
        branch_1 = slim.conv2d(branch_1, 192, [1, 7],
                              scope='Conv2d_0b_1x7')
        branch_1 = slim.conv2d(branch_1, 192, [7, 1],
                              scope='Conv2d_0c_7x1')
    with tf.variable_scope('Branch_2'):
        branch_2 = slim.conv2d(net, 192, [1, 1], scope='Conv2d_0a_1x1')
        branch_2 = slim.conv2d(branch_2, 192, [7, 1],
                              scope='Conv2d_0b_7x1')

```

```

branch_2 = slim.conv2d(branch_2, 192, [1, 7],
                      scope='Conv2d_0c_1x7')
branch_2 = slim.conv2d(branch_2, 192, [7, 1],
                      scope='Conv2d_0d_7x1')
branch_2 = slim.conv2d(branch_2, 192, [1, 7],
                      scope='Conv2d_0e_1x7')

with tf.variable_scope('Branch_3'):
    branch_3 = slim.avg_pool2d(net, [3, 3], scope='AvgPool_0a_3x3')
    branch_3 = slim.conv2d(branch_3, 192, [1, 1],
                          scope='Conv2d_0b_1x1')

net = tf.concat([branch_0, branch_1, branch_2, branch_3], 3)
end_points['Mixed_6e'] = net

```

第3个Inception模块组包含了3个Inception Module，其中后两个Inception Module的结构非常类似，它们的结构如图6-13中第三幅图所示。其中第1个Inception Module的名称为Mixed_7a，包含了3个分支。第1个分支是192输出通道的 1×1 卷积，再接320输出通道数的 3×3 卷积，不过步长为2，padding模式为VALID，因此图片尺寸缩小为 8×8 ；第2个分支有4个卷积层，分别是192输出通道的 1×1 卷积、192输出通道的 1×7 卷积、192输出通道的 7×1 卷积，以及192输出通道的 3×3 卷积。注意最后一个卷积层同样步长为2，padding为VALID，因此最后输出的tensor尺寸为 $8\times 8\times 192$ ；第3个分支则是一个 3×3 的最大池化层，步长为2，padding为VALID，而池化层不会对输出通道产生改变，因此这个分支的输出尺寸为 $8\times 8\times 768$ 。最后，我们将3个分支在输出通道上合并，输出tensor尺寸为 $8\times 8\times (320+192+768)=8\times 8\times 1280$ 。从这个Inception Module开始，输出的图片尺寸又被缩小了，同时通道数也增加了，tensor的总size在持续下降中。

```

with tf.variable_scope('Mixed_7a'):
    with tf.variable_scope('Branch_0'):
        branch_0 = slim.conv2d(net, 192, [1, 1], scope='Conv2d_0a_1x1')
        branch_0 = slim.conv2d(branch_0, 320, [3, 3], stride=2,
                              padding='VALID', scope='Conv2d_1a_3x3')

    with tf.variable_scope('Branch_1'):
        branch_1 = slim.conv2d(net, 192, [1, 1], scope='Conv2d_0a_1x1')
        branch_1 = slim.conv2d(branch_1, 192, [1, 7],
                              scope='Conv2d_0b_1x7')

```

```

branch_1 = slim.conv2d(branch_1, 192, [7, 1],
                      scope='Conv2d_0c_7x1')
branch_1 = slim.conv2d(branch_1, 192, [3, 3], stride=2,
                      padding='VALID', scope='Conv2d_1a_3x3')
with tf.variable_scope('Branch_2'):
    branch_2 = slim.max_pool2d(net, [3, 3], stride=2, padding='VALID',
                               scope='MaxPool_1a_3x3')
net = tf.concat([branch_0, branch_1, branch_2], 3)

```

接下来是第 3 个 Inception 模块组的第 2 个 Inception Module，它有 4 个分支。第 1 个分支是一个简单的 320 输出通道的 1×1 卷积；第 2 个分支先是 1 个 384 输出通道的 1×1 卷积，随后在分支内开了两个分支，这两个分支分别是 384 输出通道的 1×3 卷积和 384 输出通道的 3×1 卷积，然后使用 `tf.concat` 合并两个分支，得到的输出 tensor 尺寸为 $8 \times 8 \times (384+384) = 8 \times 8 \times 768$ ；第 3 个分支更复杂，先是 448 输出通道的 1×1 卷积，然后是 384 输出通道的 3×3 卷积，然后同样在分支内拆成两个分支，分别是 384 输出通道的 1×3 卷积和 384 输出通道的 3×1 卷积，最后合并得到 $8 \times 8 \times 768$ 的输出 tensor；第 4 个分支是在一个 3×3 的平均池化层后接一个 192 输出通道的 1×1 卷积。最后，将这个非常复杂的 Inception Module 的 4 个分支合并在一起，得到的输出 tensor 尺寸为 $8 \times 8 \times (320+768+768+192) = 8 \times 8 \times 2048$ 。到这个 Inception Module，输出通道数从 1280 增加到了 2048。

```

with tf.variable_scope('Mixed_7b'):
    with tf.variable_scope('Branch_0'):
        branch_0 = slim.conv2d(net, 320, [1, 1], scope='Conv2d_0a_1x1')
    with tf.variable_scope('Branch_1'):
        branch_1 = slim.conv2d(net, 384, [1, 1], scope='Conv2d_0a_1x1')
        branch_1 = tf.concat([
            slim.conv2d(branch_1, 384, [1, 3], scope='Conv2d_0b_1x3'),
            slim.conv2d(branch_1, 384, [3, 1], scope='Conv2d_0b_3x1')], 3)
    with tf.variable_scope('Branch_2'):
        branch_2 = slim.conv2d(net, 448, [1, 1], scope='Conv2d_0a_1x1')
        branch_2 = slim.conv2d(branch_2, 384, [3, 3],
                              scope='Conv2d_0b_3x3')
        branch_2 = tf.concat([

```

```

        slim.conv2d(branch_2, 384, [1, 3], scope='Conv2d_0c_1x3'),
        slim.conv2d(branch_2, 384, [3, 1], scope='Conv2d_0d_3x1')], 3)
with tf.variable_scope('Branch_3'):
    branch_3 = slim.avg_pool2d(net, [3, 3], scope='AvgPool_0a_3x3')
    branch_3 = slim.conv2d(branch_3, 192, [1, 1],
                          scope='Conv2d_0b_1x1')
net = tf.concat([branch_0, branch_1, branch_2, branch_3], 3)

```

Mixed_7c 是第 3 个 Inception 模块组的最后一个 Inception Module，不过它和前面的 Mixed_7b 是完全一致的，输出 tensor 也是 $8 \times 8 \times 2048$ 。最后，我们返回这个 Inception Module 的结果，作为 inception_v3_base 函数的最终输出。

```

with tf.variable_scope('Mixed_7c'):
    with tf.variable_scope('Branch_0'):
        branch_0 = slim.conv2d(net, 320, [1, 1], scope='Conv2d_0a_1x1')
    with tf.variable_scope('Branch_1'):
        branch_1 = slim.conv2d(net, 384, [1, 1], scope='Conv2d_0a_1x1')
        branch_1 = tf.concat([
            slim.conv2d(branch_1, 384, [1, 3], scope='Conv2d_0b_1x3'),
            slim.conv2d(branch_1, 384, [3, 1], scope='Conv2d_0c_3x1')], 3)
    with tf.variable_scope('Branch_2'):
        branch_2 = slim.conv2d(net, 448, [1, 1], scope='Conv2d_0a_1x1')
        branch_2 = slim.conv2d(branch_2, 384, [3, 3],
                              scope='Conv2d_0b_3x3')
        branch_2 = tf.concat([
            slim.conv2d(branch_2, 384, [1, 3], scope='Conv2d_0c_1x3'),
            slim.conv2d(branch_2, 384, [3, 1], scope='Conv2d_0d_3x1')], 3)
    with tf.variable_scope('Branch_3'):
        branch_3 = slim.avg_pool2d(net, [3, 3], scope='AvgPool_0a_3x3')
        branch_3 = slim.conv2d(branch_3, 192, [1, 1],
                              scope='Conv2d_0b_1x1')
net = tf.concat([branch_0, branch_1, branch_2, branch_3], 3)
return net, end_points

```

至此，Inception V3 网络的核心部分，即卷积层部分就完成了。回忆一下 Inception V3

的网络结构：首先是 5 个卷积层和 2 个池化层交替的普通结构，然后是 3 个 Inception 模块组，每个模块组内包含多个结构类似的 Inception Module。设计 Inception Net 的一个重要原则是，图片尺寸是不断缩小的，从 299×299 通过 5 个步长为 2 的卷积层或池化层后，缩小为 8×8 ；同时，输出通道数持续增加，从一开始的 3 (RGB 三色) 到 2048。从这里可以看出，每一层卷积、池化或 Inception 模块组的目的都是将空间结构简化，同时将空间信息转化为高阶抽象的特征信息，即将空间的维度转为通道的维度。这一过程同时也使每层输出 tensor 的总 size 持续下降，降低了计算量。读者可能也发现了 Inception Module 的规律，一般情况下有 4 个分支，第 1 个分支一般是 1×1 卷积，第 2 个分支一般是 1×1 卷积再接分解后 (factorized) 的 $1 \times n$ 和 $n \times 1$ 卷积，第 3 个分支和第 2 个分支类似，但是一般更深一些，第 4 个分支一般具有最大池化或平均池化。因此，Inception Module 是通过组合比较简单的特征抽象 (分支 1)、比较复杂的特征抽象 (分支 2 和分支 3) 和一个简化结构的池化层 (分支 4)，一共 4 种不同程度的特征抽象和变换来有选择地保留不同层次的高阶特征，这样可以最大程度地丰富网络的表达能力。

接下来，我们来实现 Inception V3 网络的最后一部分——全局平均池化、Softmax 和 Auxiliary Logits。先看函数 `inception_v3` 的输入参数，`num_classes` 即最后需要分类的数量，这里默认的 1000 是 ILSVRC 比赛数据集的种类数；`is_training` 标志是否是训练过程，对 Batch Normalization 和 Dropout 有影响，只有在训练时 Batch Normalization 和 Dropout 才会被启用；`dropout_keep_prob` 即训练时 Dropout 所需保留节点的比例，默认为 0.8；`prediction_fn` 是最后用来进行分类的函数，这里默认是使用 `slim.softmax`；`spatial_squeeze` 参数标志是否对输出进行 `squeeze` 操作 (即去除维数为 1 的维度，比如 $5 \times 3 \times 1$ 转为 5×3)；`reuse` 标志是否会对网络和 Variable 进行重复使用；最后，`scope` 为包含了函数默认参数的环境。首先，使用 `tf.variable_scope` 定义网络的 name 和 reuse 等参数的默认值，然后使用 `slim.arg_scope` 定义 Batch Normalization 和 Dropout 的 `is_training` 标志的默认值。最后，使用前面定义好的 `inception_v3_base` 构筑整个网络的卷积部分，拿到最后一层的输出 `net` 和重要节点的字典表 `end_points`。

```
def inception_v3(inputs,
                  num_classes=1000,
                  is_training=True,
                  dropout_keep_prob=0.8,
                  prediction_fn=slim.softmax,
                  spatial_squeeze=True,
```

```

        reuse=None,
        scope='InceptionV3'):

with tf.variable_scope(scope, 'InceptionV3', [inputs, num_classes],
                      reuse=reuse) as scope:
    with slim.arg_scope([slim.batch_norm, slim.dropout],
                        is_training=is_training):
        net, end_points = inception_v3_base(inputs, scope=scope)

```

接下来处理 Auxiliary Logits 这部分的逻辑，Auxiliary Logits 作为辅助分类的节点，对分类结果预测有很大帮助。先使用 `slim.arg_scope` 将卷积、最大池化、平均池化的默认步长设为 1，默认 padding 模式设为 SAME。然后通过 `end_points` 取到 Mixed_6e，并在 Mixed_6e 之后再接一个 5×5 的平均池化，步长为 3，padding 设为 VALID，这样输出的尺寸就从 17×17×768 变为 5×5×768。接着连接一个 128 输出通道的 1×1 卷积和一个 768 输出通道的 5×5 卷积，这里权重初始化方式重设为标准差为 0.01 的正态分布，padding 模式设为 VALID，输出尺寸变为 1×1×768。然后再连接一个输出通道数为 `num_classes` 的 1×1 卷积，不设激活函数和规范化函数，权重初始化方式重设为标准差为 0.001 的正态分布，这样输出变为了 1×1×1000。接下来，使用 `tf.squeeze` 函数消除输出 tensor 中前两个为 1 的维度。最后将辅助分类节点的输出 `aux_logits` 储存到字典表 `end_points` 中。

```

with slim.arg_scope([slim.conv2d, slim.max_pool2d, slim.avg_pool2d],
                    stride=1, padding='SAME'):
    aux_logits = end_points['Mixed_6e']
with tf.variable_scope('AuxLogits'):
    aux_logits = slim.avg_pool2d(
        aux_logits, [5, 5], stride=3, padding='VALID',
        scope='AvgPool_1a_5x5')
    aux_logits = slim.conv2d(aux_logits, 128, [1, 1],
                           scope='Conv2d_1b_1x1')

    aux_logits = slim.conv2d(
        aux_logits, 768, [5, 5],
        weights_initializer=trunc_normal(0.01),
        padding='VALID', scope='Conv2d_2a_5x5')

```

```

aux_logits = slim.conv2d(
    aux_logits, num_classes, [1, 1], activation_fn=None,
    normalizer_fn=None, weights_initializer=trunc_normal(0.001),
    scope='Conv2d_2b_1x1')

if spatial_squeeze:
    aux_logits = tf.squeeze(aux_logits, [1, 2],
                           name='SpatialSqueeze')

end_points['AuxLogits'] = aux_logits

```

下面处理正常的分类预测的逻辑。我们直接对 Mixed_7e 即最后一个卷积层的输出进行一个 8×8 全局平均池化，padding 模式为 VALID，这样输出 tensor 的尺寸就变为了 $1 \times 1 \times 2048$ 。然后连接一个 Dropout 层，节点保留率为 dropout_keep_prob。接着连接一个输出通道数为 1000 的 1×1 卷积，激活函数和规范化函数设为空。下面使用 tf.squeeze 去除输出 tensor 中维数为 1 的维度，再连接一个 Softmax 对结果进行分类预测。最后返回输出结果 logits 和包含辅助节点的 end_points。

```

with tf.variable_scope('Logits'):
    net = slim.avg_pool2d(net, [8, 8], padding='VALID',
                          scope='AvgPool_1a_8x8')
    net = slim.dropout(net, keep_prob=dropout_keep_prob,
                      scope='Dropout_1b')
    end_points['PreLogits'] = net
    logits = slim.conv2d(net, num_classes, [1, 1], activation_fn=None,
                        normalizer_fn=None, scope='Conv2d_1c_1x1')
    if spatial_squeeze:
        logits = tf.squeeze(logits, [1, 2], name='SpatialSqueeze')
    end_points['Logits'] = logits
    end_points['Predictions'] = prediction_fn(logits, scope='Predictions')
return logits, end_points

```

至此，整个 Inception V3 网络的构建就完成了。Inception V3 是一个非常复杂、精妙的模型，其中用到了非常多之前积累下来的设计大型卷积网络的经验和技巧。不过，虽然 Inception V3 论文中给出了设计卷积网络的几个原则，但是其中很多超参数的选择，包括层数、卷积核的尺寸、池化的位置、步长的大小、factorization 使用的时机，以及分支的

设计，都很难一一解释。目前，我们只能认为深度学习，尤其是大型卷积网络的设计，是一门实验学科，其中需要大量的探索和实践。我们很难证明某种网络结构一定更好，更多的是通过实验积累下来的经验总结出一些结论。深度学习的研究中，理论证明部分依然是短板，但通过实验得到的结论通常也具有不错的推广性，在其他数据集上泛化性良好。

下面对 Inception V3 进行运算性能测试。这里使用的 time_tensorflow_run 函数和 AlexNet 那节一样，因此就不再重复定义，读者可以在 6.1 节中找到代码并加载。因为 Inception V3 网络结构较大，所以依然令 batch_size 为 32，以免 GPU 显存不够。图片尺寸设置为 299×299 ，并用 tf.random_uniform 生成随机图片数据作为 input。接着，我们使用 slim.arg_scope 加载前面定义好的 inception_v3_arg_scope()，在这个 scope 中包含了 Batch Normalization 的默认参数，以及激活函数和参数初始化方式的默认值。然后在这个 arg_scope 下，调用 inception_v3 函数，并传入 inputs，获取 logits 和 end_points。下面创建 Session 并初始化全部模型参数。最后我们设置测试的 batch 数量为 100，并使用 time_tensorflow_run 测试 Inception V3 网络的 forward 性能。

```
batch_size = 32
height, width = 299, 299
inputs = tf.random_uniform((batch_size, height, width, 3))
with slim.arg_scope(inception_v3_arg_scope()):
    logits, end_points = inception_v3(inputs, is_training=False)

init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)
num_batches=100
time_tensorflow_run(sess, logits, "Forward")
```

从结果来看，Inception V3 网络的 forward 性能不错，在 GTX 1080、CUDA 8、cuDNN 5.1 的环境下，每个 batch（包含 32 张图片）预测耗时仅为 0.145s。虽然输入图片的面积比 VGGNet 的 224×224 大了 78%，但是 forward 速度却比 VGGNet 的 0.152s 更快。这主要归功于其较小的参数量，Inception V3 网络仅有 2500 万个参数，虽然比 Inception V1 的 700 万多了很多，不过仍然不到 AlexNet 的 6000 万参数量的一半，相比 VGGNet 的 1.4 亿参数量就更少了，这对一个 42 层深的大型网络来说是极为不易的。同时，整个网络的浮点计算量仅为 50 亿次，虽也比 Inception V1 的 15 亿次大了不少，但是相比 VGGNet

仍然不算大。较小的计算量让 Inception V3 网络变得非常实用，我们可以轻松地将其移植到普通服务器上提供快速响应的服务，甚至是移植到手机上进行实时的图像识别。

```
2016-12-10 21:07:09.535980: step 0, duration = 0.145
2016-12-10 21:07:10.982748: step 10, duration = 0.145
2016-12-10 21:07:12.430209: step 20, duration = 0.145
2016-12-10 21:07:13.877055: step 30, duration = 0.145
2016-12-10 21:07:15.324095: step 40, duration = 0.145
2016-12-10 21:07:16.770960: step 50, duration = 0.145
2016-12-10 21:07:18.218127: step 60, duration = 0.145
2016-12-10 21:07:19.665192: step 70, duration = 0.145
2016-12-10 21:07:21.113429: step 80, duration = 0.145
2016-12-10 21:07:22.563213: step 90, duration = 0.145
2016-12-10 21:07:23.867730: Forward across 100 steps, 0.145 +/- 0.000 sec /
batch
```

因为篇幅原因，我们就不对 Inception V3 的 backward 性能进行测试了，这部分的代码比较冗长。感兴趣的读者，可以将整个网络的所有参数加入参数列表，测试对全部参数求导所需的时间，或者直接下载 ImageNet 数据集，使用真实样本进行训练并评测所需时间。

Inception V3 作为一个极深的卷积神经网络，拥有非常精妙的设计和构造，整个网络的结构和分支非常复杂。我们平时可能不必设计这么复杂的网络，但 Inception V3 中仍有许多设计 CNN 的思想和 Trick 值得借鉴。

(1) Factorization into small convolutions 很有效，可以降低参数量、减轻轻过拟合，增加网络非线性的表达能力。

(2) 卷积网络从输入到输出，应该让图片尺寸逐渐减小，输出通道数逐渐增加，即让空间结构简化，将空间信息转化为高阶抽象的特征信息。

(3) Inception Module 用多个分支提取不同抽象程度的高阶特征的思路很有效，可以丰富网络的表达能力。

6.4 TensorFlow 实现 ResNet

ResNet (Residual Neural Network) 由微软研究院的 Kaiming He 等 4 名华人提出，通过使用 Residual Unit 成功训练 152 层深的神经网络，在 ILSVRC 2015 比赛中获得了冠军，取得 3.57% 的 top-5 错误率，同时参数量却比 VGGNet 低，效果非常突出。ResNet 的结构可以极快地加速超深神经网络的训练，模型的准确率也有非常大的提升。6.3 节我们讲解并实现了 Inception V3，而 Inception V4 则是将 Inception Module 和 ResNet 相结合。可以看到 ResNet 是一个推广性非常好的网络结构，甚至可以直接应用到 Inception Net 中。本节就讲解 ResNet 的基本原理，以及如何用 TensorFlow 来实现它。

在 ResNet 之前，瑞士教授 Schmidhuber 提出了 Highway Network，原理与 ResNet 很相似。这位 Schmidhuber 教授同时也是 LSTM 网络的发明者，而且是早在 1997 年发明的，可谓是神经网络领域元老级的学者。通常认为神经网络的深度对其性能非常重要，但是网络越深其训练难度越大，Highway Network 的目标就是解决极深的神经网络难以训练的问题。Highway Network 相当于修改了每一层的激活函数，此前的激活函数只是对输入做一个非线性变换 $y = H(\mathbf{x}, \mathbf{W}_H)$ ，Highway Network 则允许保留一定比例的原始输入 \mathbf{x} ，即 $y = H(\mathbf{x}, \mathbf{W}_H) \cdot T(\mathbf{x}, \mathbf{W}_T) + \mathbf{x} \cdot C(\mathbf{x}, \mathbf{W}_C)$ ，其中 T 为变换系数， C 为保留系数，论文中令 $C = 1 - T$ 。这样前面一层的信息，有一定比例可以不经过矩阵乘法和非线性变换，直接传输到下一层，仿佛一条信息高速公路，因此得名 Highway Network。Highway Network 主要通过 gating units 学习如何控制网络中的信息流，即学习原始信息应保留的比例。这个可学习的 gating 机制，正是借鉴自 Schmidhuber 教授早年的 LSTM 循环神经网络中的 gating。几百乃至上千层深的 Highway Network 可以直接使用梯度下降算法训练，并可以配合多种非线性激活函数，学习极深的神经网络现在变得可行了。事实上，Highway Network 的设计在理论上允许其训练任意深的网络，其优化方法基本上与网络的深度独立，而传统的神经网络结构则对深度非常敏感，训练复杂度随深度增加而急剧增加。

ResNet 和 Highway Network 非常类似，也是允许原始输入信息直接传输到后面的层中。ResNet 最初的灵感出自这个问题：在不断加神经网络的深度时，会出现一个 Degradation 的问题，即准确率会先上升然后达到饱和，再持续增加深度则会导致准确率下降。这并不是过拟合的问题，因为不光在测试集上误差增大，训练集本身误差也会增大。假设有一个比较浅的网络达到了饱和的准确率，那么后面再加上几个 $y = \mathbf{x}$ 的全等映射层，起码误差不会增加，即更深的网络不应该带来训练集上误差上升。而这里提到的使用全等

映射直接将前一层输出传到后面的思想，就是 ResNet 的灵感来源。假定某段神经网络的输入是 x ，期望输出是 $H(x)$ ，如果我们直接把输入 x 传到输出作为初始结果，那么此时我们需要学习的目标就是 $F(x) = H(x) - x$ 。如图 6-14 所示，这就是一个 ResNet 的残差学习单元(Residual Unit)，ResNet 相当于将学习目标改变了，不再是学习一个完整的输出 $H(x)$ ，只是输出和输入的差别 $H(x)-x$ ，即残差。

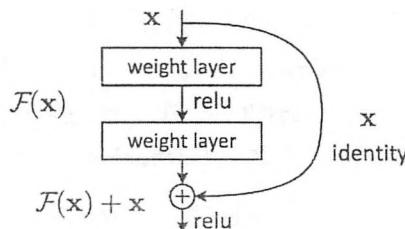


图 6-14 ResNet 的残差学习模块

图 6-15 所示为 VGGNet-19，以及一个 34 层深的普通卷积网络，和 34 层深的 ResNet 网络的对比图。可以看到普通直连的卷积神经网络和 ResNet 的最大区别在于，ResNet 有很多旁路的支线将输入直接连到后面的层，使得后面的层可以直接学习残差，这种结构也被称为 shortcut 或 skip connections。

传统的卷积层或全连接层在信息传递时，或多或少会存在信息丢失、损耗等问题。ResNet 在某种程度上解决了这个问题，通过直接将输入信息绕道传到输出，保护信息的完整性，整个网络则只需要学习输入、输出差别的那一部分，简化学习目标和难度。

在 ResNet 的论文中，除了提出图 6-16 中的两层残差学习单元，还有三层的残差学习单元。两层的残差学习单元中包含两个相同输出通道数(因为残差等于目标输出减去输入，即 $H(x)-x$ ，因此输入、输出维度需保持一致)的 3×3 卷积；而 3 层的残差网络则使用了 Network In Network 和 Inception Net 中的 1×1 卷积，并且是在中间 3×3 的卷积前后都使用了 1×1 卷积，有先降维再升维的操作。另外，如果有输入、输出维度不同的情况，我们可以对 x 做一个线性映射变换维度，再连接到后面的层。

图 6-17 所示为 ResNet 在不同层数时的网络配置，其中基础结构很类似，都是前面提到的两层和三层的残差学习单元的堆叠。

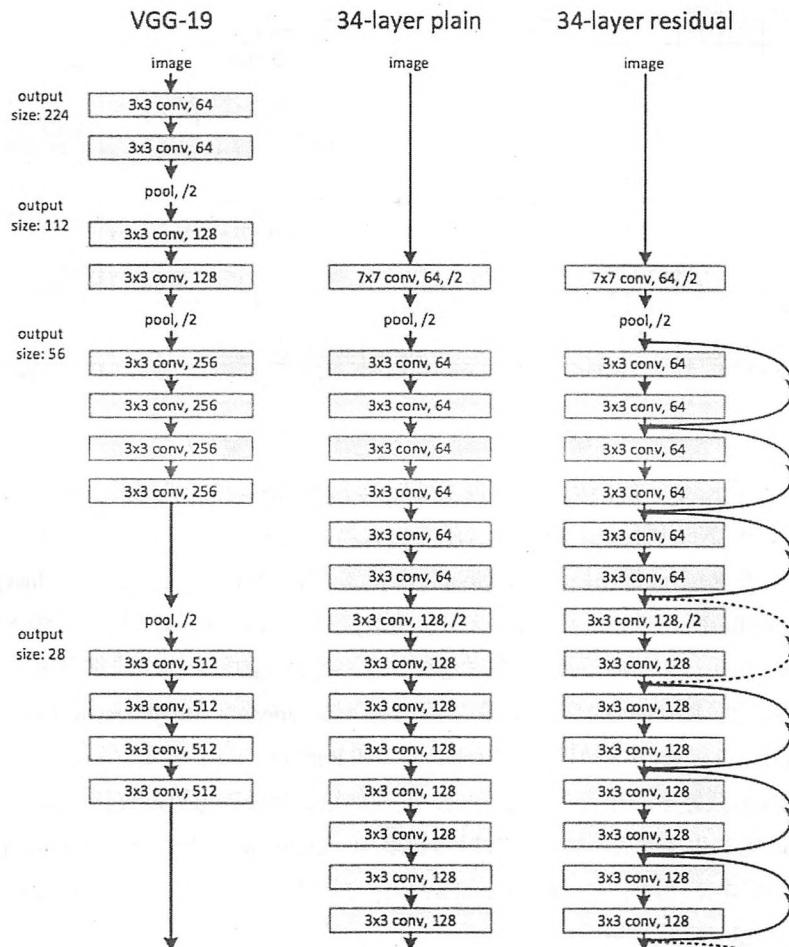


图 6-15 VGG-19, 直连的 34 层网络, ResNet 的 34 层网络的结构对比

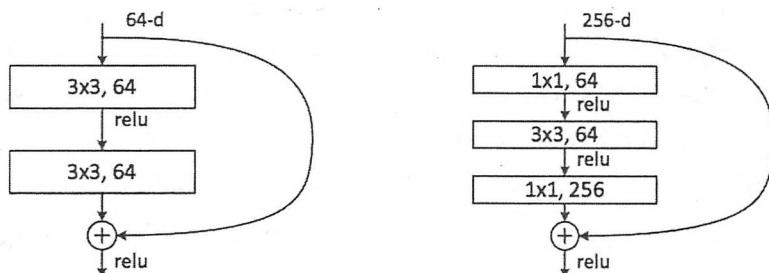


图 6-16 两层及三层的 ResNet 残差学习模块

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112			7×7, 64, stride 2		
				3×3 max pool, stride 2		
conv2_x	56×56	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$
conv3_x	28×28	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 8$
conv4_x	14×14	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 36$
conv5_x	7×7	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$
	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

图 6-17 ResNet 不同层数时的网络配置

在使用了 ResNet 的结构后，可以发现层数不断加深导致的训练集上误差增大的现象被消除了，ResNet 网络的训练误差会随着层数增大而逐渐减小，并且在测试集上的表现也会变好。在 ResNet 推出后不久，Google 就借鉴了 ResNet 的精髓，提出了 Inception V4 和 Inception-ResNet-V2，并通过融合这两个模型，在 ILSVRC 数据集上取得了惊人的 3.08% 的错误率。可见，ResNet 及其思想对卷积神经网络研究的贡献确实非常显著，具有很强的推广性。在 ResNet 的作者的第二篇相关论文 *Identity Mappings in Deep Residual Networks* 中，ResNet V2 被提出。ResNet V2 和 ResNet V1 的主要区别在于，作者通过研究 ResNet 残差学习单元的传播公式，发现前馈和反馈信号可以直接传输，因此 skip connection 的非线性激活函数(如 ReLU)替换为 Identity Mappings($y = x$)。同时，ResNet V2 在每一层中都使用了 Batch Normalization。这样处理之后，新的残差学习单元将比以前更容易训练且泛化性更强。

根据 Schmidhuber 教授的观点，ResNet 类似于一个没有 gates 的 LSTM 网络，即将输入 x 传递到后面层的过程是一直发生的，而不是学习出来的。同时，最近也有两篇论文表示，ResNet 基本等价于 RNN 且 ResNet 的效果类似于在多层网络间的集成方法(ensemble)。ResNet 在加深网络层数上做出了重大贡献，而另一篇论文 *The Power of Depth for Feedforward Neural Networks* 则从理论上证明了加深网络比加宽网络更有效，算是给 ResNet 提供了声援，也是给深度学习为什么要深才有效提供了合理解释。

下面我们就用 TensorFlow 实现一个 ResNet V2 网络。我们依然使用方便的 contrib.slim 库来辅助创建 ResNet，其余载入的库还有原生的 collections。本节代码主要来自 TensorFlow 的开源实现⁵⁴。

```
import collections
import tensorflow as tf
slim = tf.contrib.slim
```

我们使用 collections.namedtuple 设计 ResNet 基本 Block 模块组(图 6-17 中所示的 Block) 的 named tuple，并用它创建 Block 的类，但只包含数据结构，不包含具体方法。我们要定义一个典型的 Block，需要输入三个参数，分别是 scope、unit_fn 和 args。以 Block('block1', bottleneck, [(256, 64, 1)] × 2 + [(256, 64, 2)]) 这一行代码为例，它可以定义一个典型的 Block，其中 block1 就是我们这个 Block 的名称（或 scope）；bottleneck 是 ResNet V2 中的残差学习单元；而最后一个参数 [(256, 64, 1)] × 2 + [(256, 64, 2)] 则是这个 Block 的 args，args 是一个列表，其中每个元素都对应一个 bottleneck 残差学习单元，前面两个元素都是 (256, 64, 1)，最后一个 (256, 64, 2)。每个元素都是一个三元 tuple，即 (depth, depth_bottleneck, stride)。比如 (256, 64, 3)，代表构建的 bottleneck 残差学习单元（每个残差学习单元包含三个卷积层）中，第三层输出通道数 depth 为 256，前两层输出通道数 depth_bottleneck 为 64，且中间那层的步长 stride 为 3。这个残差学习单元结构即为 [(1×1/s1, 64), (3×3/s2, 64), (1×1/s1, 256)]。而在一个 Block 中，一共有 3 个 bottleneck 残差学习单元，除了最后一个的步长由 3 变为 2，其余都一致。

```
class Block(collections.namedtuple('Block', ['scope', 'unit_fn', 'args']):
    'A named tuple describing a ResNet block.'
```

下面定义一个降采样 subsample 的方法，参数包括 inputs（输入）、factor（采样因子）和 scope。这个函数也非常简单，如果 factor 为 1，则不做修改直接返回 inputs；如果不为 1，则使用 slim.max_pool2d 最大池化来实现，通过 1×1 的池化尺寸，stride 作步长，即可实现降采样。

```
def subsample(inputs, factor, scope=None):
    if factor == 1:
        return inputs
    else:
        return slim.max_pool2d(inputs, [1, 1], stride=factor, scope=scope)
```

再定义一个 conv2d_same 函数创建卷积层。先判断 stride 是否为 1，如果为 1，则直接使用 slim.conv2d 并令 padding 模式为 SAME。如果 stride 不为 1，则显式地 pad zero，要 pad zero 的总数为 kernel_size-1，pad_beg 为 pad//2，pad_end 为余下的部分。接下来使

用 tf.pad 对输入变量进行补零操作。最后，因为已经进行了 zero padding，所以只需再使用一个 padding 模式为 VALID 的 slim.conv2d 创建这个卷积层。

```
def conv2d_same(inputs, num_outputs, kernel_size, stride, scope=None):
    if stride == 1:
        return slim.conv2d(inputs, num_outputs, kernel_size, stride=1,
                           padding='SAME', scope=scope)
    else:
        pad_total = kernel_size - 1
        pad_beg = pad_total // 2
        pad_end = pad_total - pad_beg
        inputs = tf.pad(inputs, [[0, 0], [pad_beg, pad_end],
                               [pad_beg, pad_end], [0, 0]])
        return slim.conv2d(inputs, num_outputs, kernel_size, stride=stride,
                           padding='VALID', scope=scope)
```

接下来定义堆叠 Blocks 的函数，参数中的 net 即为输入，blocks 是之前定义的 Block 的 class 的列表，而 outputs_collections 则是用来收集各个 end_points 的 collections。下面使用两层循环，逐个 Block，逐个 Residual Unit 地堆叠，先使用两个 tf.variable_scope 将残差学习单元命名为 block1/unit_1 的形式。在第 2 层循环中，我们拿到每个 Block 中每个 Residual Unit 的 args，并展开为 depth、depth_bottleneck 和 stride，其含义在前面定义 Blocks 类时已经讲解过。然后使用 unit_fn 函数（即残差学习单元的生成函数）顺序地创建并连接所有的残差学习单元。最后，我们使用 slim.utils.collect_named_outputs 函数将输出 net 添加到 collection 中。最后，当所有 Block 中的所有 Residual Unit 都堆叠完之后，我们再返回最后的 net 作为 stack_blocks_dense 函数的结果。

```
@slim.add_arg_scope
def stack_blocks_dense(net, blocks, outputs_collections=None):

    for block in blocks:
        with tf.variable_scope(block.scope, 'block', [net]) as sc:
            for i, unit in enumerate(block.args):
                with tf.variable_scope('unit_%d' % (i + 1), values=[net]):
                    unit_depth, unit_depth_bottleneck, unit_stride = unit
```

```

    net = block.unit_fn(net,
                         depth=unit_depth,
                         depth_bottleneck=unit_depth_bottleneck,
                         stride=unit_stride)
    net = slim.utils.collect_named_outputs(outputs_collections, sc.name,
                                           net)

return net

```

这里创建 ResNet 通用的 arg_scope，关于 arg_scope，我们在前面的章节已经介绍过其功能——用来定义某些函数的参数默认值。这里定义训练标记 is_training 默认为 True，权重衰减速率 weight_decay 默认为 0.0001，BN 的衰减速率默认为 0.997，BN 的 epsilon 默认为 $1e-5$ ，BN 的 scale 默认为 True。和在 Inception V3 定义 arg_scope 一样，先设置好 BN 的各项参数，然后通过 slim.arg_scope 将 slim.conv2d 的几个默认参数设置好：权重正则器设置为 L2 正则，权重初始化器设为 slim.variance_scaling_initializer()，激活函数设为 ReLU，标准化器设为 BN。并将最大池化的 padding 模式默认设为 SAME（注意，ResNet 原论文中使用的是 VALID 模式，设为 SAME 可让特征对齐更简单，读者可以尝试改为 VALID）。最后，将几层嵌套的 arg_scope 作为结果返回。

```

def resnet_arg_scope(is_training=True,
                     weight_decay=0.0001,
                     batch_norm_decay=0.997,
                     batch_norm_epsilon=1e-5,
                     batch_norm_scale=True):

    batch_norm_params = {
        'is_training': is_training,
        'decay': batch_norm_decay,
        'epsilon': batch_norm_epsilon,
        'scale': batch_norm_scale,
        'updates_collections': tf.GraphKeys.UPDATE_OPS,
    }

    with slim.arg_scope(

```

```
[slim.conv2d],
weights_regularizer=slim.l2_regularizer(weight_decay),
weights_initializer=slim.variance_scaling_initializer(),
activation_fn=tf.nn.relu,
normalizer_fn=slim.batch_norm,
normalizer_params=batch_norm_params):
with slim.arg_scope([slim.batch_norm], **batch_norm_params):
    with slim.arg_scope([slim.max_pool2d], padding='SAME') as arg_sc:
        return arg_sc
```

接下来定义核心的 bottleneck 残差学习单元，它是 ResNet V2 的论文中提到的 Full Preactivation Residual Unit 的一个变种。它和 ResNet V1 中的残差学习单元的主要区别有两点，一是在每一层前都用了 Batch Normalization，二是对输入进行 preactivation，而不是在卷积进行激活函数处理。我们来看一下 bottleneck 函数的参数，inputs 是输入，depth、depth_bottleneck 和 stride 这三个参数前面的 Blocks 类中的 args，outputs_collections 是收集 end_points 的 collection，scope 是这个 unit 的名称。下面先使用 slim.utils.last_dimension 函数获取输入的最后一个维度，即输出通道数，其中的参数 min_rank=4 可以限定最少为 4 个维度。接着，使用 slim.batch_norm 对输入进行 Batch Normalization，并使用 ReLU 函数进行预激活 Preactivate。然后定义 shorcut（即直连的 x）：如果残差单元的输入通道数 depth_in 和输出通道数 depth 一致，那么使用 subsample 按步长为 stride 对 inputs 进行空间上的降采样（确保空间尺寸和残差一致，因为残差中间那层的卷积步长为 stride）；如果输入、输出通道数不一样，我们用步长为 stride 的 1×1 卷积改变其通道数，使得与输出通道数一致。然后定义 residual（残差），residual 这里有 3 层，先是一个 1×1 尺寸、步长为 1、输出通道数为 depth_bottleneck 的卷积，然后是一个 3×3 尺寸、步长为 stride、输出通道数为 depth_bottleneck 的卷积，最后是一个 1×1 卷积、步长为 1、输出通道数为 depth 的卷积，得到最终的 residual，这里注意最后一层没有正则项也没有激活函数。然后将 residual 和 shorcut 相加，得到最后结果 output，再使用 slim.utils.collect_named_outputs 将结果添加进 collection 并返回 output 作为函数结果。

```
@slim.add_arg_scope
def bottleneck(inputs, depth, depth_bottleneck, stride,
               outputs_collections=None, scope=None):
```

```

with tf.variable_scope(scope, 'bottleneck_v2', [inputs]) as sc:
    depth_in = slim.utils.last_dimension(inputs.get_shape(), min_rank=4)
    preact = slim.batch_norm(inputs, activation_fn=tf.nn.relu,
                            scope='preact')
    if depth == depth_in:
        shortcut = subsample(inputs, stride, 'shortcut')
    else:
        shortcut = slim.conv2d(preact, depth, [1, 1], stride=stride,
                              normalizer_fn=None, activation_fn=None,
                              scope='shortcut')

    residual = slim.conv2d(preact, depth_bottleneck, [1, 1], stride=1,
                          scope='conv1')
    residual = conv2d_same(residual, depth_bottleneck, 3, stride,
                          scope='conv2')
    residual = slim.conv2d(residual, depth, [1, 1], stride=1,
                          normalizer_fn=None, activation_fn=None,
                          scope='conv3')

    output = shortcut + residual

return slim.utils.collect_named_outputs(outputs_collections,
                                       sc.name, output)

```

下面定义生成 ResNet V2 的主函数，我们只要预先定义好网络的残差学习模块组 blocks，它就可以生成对应的完整的 ResNet。先来看一下这个函数的参数，inputs 即输入，blocks 为定义好的 Block 类的列表，num_classes 是最后输出的类数，global_pool 标志是否加上最后的一层全局平均池化，include_root_block 标志是否加上 ResNet 网络最前面通常使用的 7×7 卷积和最大池化，reuse 标志是否重用，scope 是整个网络的名称。在函数体内，我们先定义好 variable_scope 及 end_points_collection，再通过 slim.arg_scope 将 (slim.conv2d、bottleneck、stack_block_dense) 这三个函数的参数 outputs_collections 默认设为 end_points_collection。然后根据 include_root_block 标记，创建 ResNet 最前面的 64 输出通道的步长为 2 的 7×7 卷积，然后再接一个步长为 2 的 3×3 最大池化。经历两个步长

为 2 的层，图片尺寸已经被缩小为 1/4。然后，使用前面定义的 `stack_blocks_dense` 将残差学习模块组生成好，再根据标记添加全局平均池化层，这里用 `tf.reduce_mean` 实现全局平均池化，效率比直接用 `avg_pool` 高。下面根据是否有分类数，添加一个输出通道为 `num_classes` 的 1×1 卷积（该卷积层无激活函数和正则项），再添加一个 `Softmax` 层输出网络结果。同时使用 `slim.utils.convert_collection_to_dict` 将 `collection` 转化为 Python 的 `dict`，最后返回 `net` 和 `end_points`。

```
def resnet_v2(inputs,
              blocks,
              num_classes=None,
              global_pool=True,
              include_root_block=True,
              reuse=None,
              scope=None):

    with tf.variable_scope(scope, 'resnet_v2', [inputs], reuse=reuse) as sc:
        end_points_collection = sc.original_name_scope + '_end_points'
        with slim.arg_scope([slim.conv2d, bottleneck,
                            stack_blocks_dense],
                            outputs_collections=end_points_collection):
            net = inputs
            if include_root_block:
                with slim.arg_scope([slim.conv2d], activation_fn=None,
                                    normalizer_fn=None):
                    net = conv2d_same(net, 64, 7, stride=2, scope='conv1')
                    net = slim.max_pool2d(net, [3, 3], stride=2, scope='pool1')
            net = stack_blocks_dense(net, blocks)
            net = slim.batch_norm(net, activation_fn=tf.nn.relu, scope='postnorm')
            if global_pool:
                net = tf.reduce_mean(net, [1, 2], name='pool5', keep_dims=True)
            if num_classes is not None:
                net = slim.conv2d(net, num_classes, [1, 1], activation_fn=None,
                                 normalizer_fn=None, scope='logits')
```

```

end_points = slim.utils.convert_collection_to_dict(
    end_points_collection)
if num_classes is not None:
    end_points['predictions'] = slim.softmax(net, scope='predictions')
return net, end_points

```

至此，我们就将 ResNet 的生成函数定义好了，下面根据图 6-17 中推荐的几个不同深度的 ResNet 网络配置，来设计层数分别为 50、101、152 和 200 的 ResNet。我们先来看 50 层的 ResNet，其严格遵守了图 6-17 所示的设置，4 个残差学习 Blocks 的 units 数量分别为 3、4、6 和 3，总层数即为 $(3+4+6+3) \times 3 + 2 = 50$ 。需要注意的是，残差学习模块之前的卷积、池化已经将尺寸缩小了 4 倍，我们前 3 个 Blocks 又都包含步长为 2 的层，因此总尺寸缩小了 $4 \times 8 = 32$ 倍，输入图片尺寸最后变为 $224/32=7$ 。和 Inception V3 很像，ResNet 不断使用步长为 2 的层来缩减尺寸，但同时输出通道数也在持续增加，最后达到了 2048。

```

def resnet_v2_50(inputs,
                  num_classes=None,
                  global_pool=True,
                  reuse=None,
                  scope='resnet_v2_50'):
    blocks = [
        Block('block1', bottleneck, [(256, 64, 1)] * 2 + [(256, 64, 2)]),
        Block('block2', bottleneck, [(512, 128, 1)] * 3 + [(512, 128, 2)]),
        Block('block3', bottleneck, [(1024, 256, 1)] * 5 + [(1024, 256, 2)]),
        Block('block4', bottleneck, [(2048, 512, 1)] * 3)]
    return resnet_v2(inputs, blocks, num_classes, global_pool,
                     include_root_block=True, reuse=reuse, scope=scope)

```

101 层的 ResNet 和 50 层相比，主要变化就是把 4 个 Blocks 的 units 数量从 3、4、6、3 提升到了 3、4、23、3。即将第三个残差学习 Block 的 units 数增加到接近 4 倍。

```

def resnet_v2_101(inputs,
                  num_classes=None,
                  global_pool=True,
                  reuse=None,

```

```

    scope='resnet_v2_101'):

blocks = [
    Block('block1', bottleneck, [(256, 64, 1)] * 2 + [(256, 64, 2)]),
    Block('block2', bottleneck, [(512, 128, 1)] * 3 + [(512, 128, 2)]),
    Block('block3', bottleneck, [(1024, 256, 1)] * 22 + [(1024, 256, 2)]),
    Block('block4', bottleneck, [(2048, 512, 1)] * 3)]
return resnet_v2(inputs, blocks, num_classes, global_pool,
                 include_root_block=True, reuse=reuse, scope=scope)

```

然后 152 层的 ResNet，则是将第二个 Block 的 units 数提高到 8，将第三个 Block 的 units 数提高到 36。Units 数量提升的主要场所依然是第三个 Block。

```

def resnet_v2_152(inputs,
                   num_classes=None,
                   global_pool=True,
                   reuse=None,
                   scope='resnet_v2_152'):

blocks = [
    Block('block1', bottleneck, [(256, 64, 1)] * 2 + [(256, 64, 2)]),
    Block('block2', bottleneck, [(512, 128, 1)] * 7 + [(512, 128, 2)]),
    Block('block3', bottleneck, [(1024, 256, 1)] * 35 + [(1024, 256, 2)]),
    Block('block4', bottleneck, [(2048, 512, 1)] * 3)]
return resnet_v2(inputs, blocks, num_classes, global_pool,
                 include_root_block=True, reuse=reuse, scope=scope)

```

最后，200 层的 ResNet 相比 152 层的 ResNet，没有继续提升第三个 Block 的 units 数，而是将第二个 Block 的 units 数一下子提升到了 23。

```

def resnet_v2_200(inputs,
                   num_classes=None,
                   global_pool=True,
                   reuse=None,
                   scope='resnet_v2_200'):

blocks = [
    Block('block1', bottleneck, [(256, 64, 1)] * 2 + [(256, 64, 2)]),

```

```

    Block('block2', bottleneck, [(512, 128, 1)] * 23 + [(512, 128, 2)]),
    Block('block3', bottleneck, [(1024, 256, 1)] * 35 + [(1024, 256, 2)]),
    Block('block4', bottleneck, [(2048, 512, 1)] * 3)]
return resnet_v2(inputs, blocks, num_classes, global_pool,
                 include_root_block=True, reuse=reuse, scope= scope)

```

最后我们使用一直以来的评测函数 time_tensorflow_run，来测试 152 层深的 ResNet（即获得 ILSVRC 2015 冠军的版本）的 forward 性能。图片尺寸回归到 AlexNet、VGGNet 的 224×224 ，batch size 为 32。我们将 is_training 这个 FLAG 置为 False，然后使用 resnet_v2_152 创建网络，再由 time_tensorflow_run 函数评测其 forward 性能。由于篇幅原因，就不对其训练时的性能进行测试了，感兴趣的读者可以测试求解 ResNet 全部参数的梯度所需要的时间。

```

batch_size = 32
height, width = 224, 224
inputs = tf.random_uniform((batch_size, height, width, 3))
with slim.arg_scope(resnet_arg_scope(is_training=False)):
    net, end_points = resnet_v2_152(inputs, 1000)

init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)
num_batches=100
time_tensorflow_run(sess, net, "Forward")

```

这里可以看到，虽然这个 ResNet 有 152 层深，但其 forward 计算耗时并没有特别夸张，相比 VGGNet 和 Inception V3，大概只增加了 50%，每 batch 为 0.202 秒。这说明 ResNet 也是一个实用的卷积神经网络结构，不仅支持超深网络的训练，同时在实际工业应用时也有不差的 forward 性能。

```

2016-12-10 21:23:43.676945: step 0, duration = 0.202
2016-12-10 21:23:45.699069: step 10, duration = 0.203
2016-12-10 21:23:47.722190: step 20, duration = 0.203
2016-12-10 21:23:49.745069: step 30, duration = 0.202
2016-12-10 21:23:51.770527: step 40, duration = 0.202

```

```
2016-12-10 21:23:53.797204: step 50, duration = 0.202
2016-12-10 21:23:55.822281: step 60, duration = 0.203
2016-12-10 21:23:57.848449: step 70, duration = 0.203
2016-12-10 21:23:59.877548: step 80, duration = 0.203
2016-12-10 21:24:01.906566: step 90, duration = 0.203
2016-12-10 21:24:03.734302: Forward across 100 steps, 0.203 +/- 0.000 sec /
batch
```

本节我们完整地讲解了 ResNet 的基本原理及其 TensorFlow 实现，也设计了一系列不同深度的 ResNet。读者若感兴趣，可以自行探索不同深度、乃至不同残差单元结构的 ResNet 的分类性能。例如，ResNet 原论文中主要增加的是第二个和第三个 Block 的 units 数，读者可以尝试增加其余两个 Block 的 units 数，或者修改 bottleneck 单元中的 depth、depth_bottleneck 等参数，可对其参数设置的意义加深理解。ResNet 可以算是深度学习中一个里程碑式的突破，真正意义上支持了极深神经网络的训练。其网络结构值得反复思索，如 Google 等已将其融合到自家的 Inception Net 中，并取得了非常好的效果。相信 ResNet 的成功也会启发其他在深度学习领域研究的灵感。

6.5 卷积神经网络发展趋势

本节，我们简单回顾卷积神经网络的历史，图 6-18 所示大致勾勒出最近几十年卷积神经网络的发展方向。Perceptron(感知机)于 1957 年由 Frank Rosenblatt 提出，而 Perceptron 不仅是卷积网络，也是神经网络的始祖。Neocognitron (神经认知机) 是一种多层级的神经网络，由日本科学家 Kunihiko Fukushima 于 20 世纪 80 年代提出，具有一定程度的视觉认知的功能，并直接启发了后来的卷积神经网络。LeNet-5 由 CNN 之父 Yann LeCun 于 1997 年提出，首次提出了多层级联的卷积结构，可对手写数字进行有效识别。可以看到前面这三次关于卷积神经网络的技术突破，间隔时间非常长，需要十余年甚至更久才出现一次理论创新。而后于 2012 年，Hinton 的学生 Alex 依靠 8 层深的卷积神经网络一举获得了 ILSVRC 2012 比赛的冠军，瞬间点燃了卷积神经网络研究的热潮。AlexNet 成功应用了 ReLU 激活函数、Dropout、最大覆盖池化、LRN 层、GPU 加速等新技术，并启发了后续更多的技术创新，卷积神经网络的研究从此进入快车道。

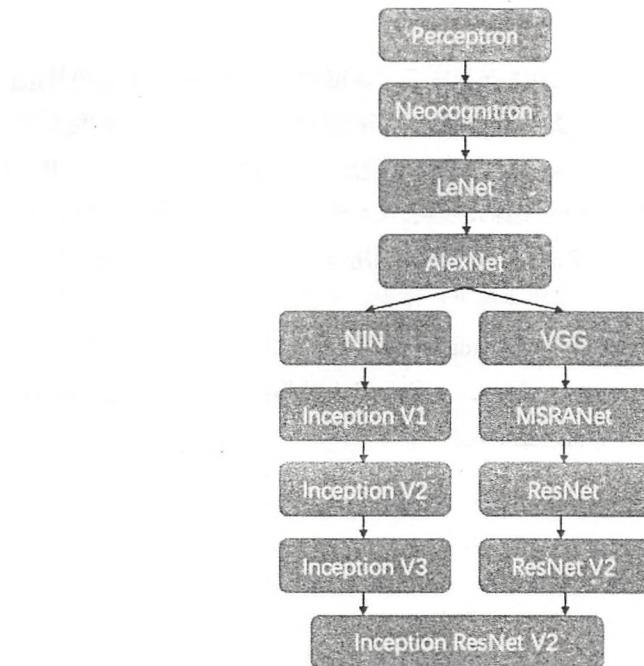
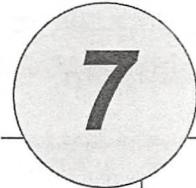


图 6-18 卷积神经网络发展图

在 AlexNet 之后，我们可以将卷积神经网络的发展分为两类，一类是网络结构上的改进调整(图 6-18 中的左侧分支)，另一类是网络深度的增加(图 6-18 中的右侧分支)。2013 年，颜水成教授的 Network in Network 工作首次发表，优化了卷积神经网络的结构，并推广了 1×1 的卷积结构。在改进卷积网络结构的工作中，后继者还有 2014 年的 Google Inception Net V1，提出了 Inception Module 这个可以反复堆叠的高效的卷积网络结构，并获得了当年 ILSVRC 比赛的冠军。2015 年初的 Inception V2 提出了 Batch Normalization，大大加速了训练过程，并提升了网络性能。2015 年年末的 Inception V3 则继续优化了网络结构，提出了 Factorization in Small Convolutions 的思想，分解大尺寸卷积为多个小卷积乃至一维卷积。而另一条分支上，许多研究工作则致力于加深网络层数，2014 年，ILSVRC 比赛的亚军 VGGNet 全程使用 3×3 的卷积，成功训练了深达 19 层的网络，当年的季军 MSRA-Net 也使用了非常深的网络。2015 年，微软的 ResNet 成功训练了 152 层深的网络，一举拿下了当年 ILSVRC 比赛的冠军，top-5 错误率降低至 3.46%。其后又更新了 ResNet V2，增加了 Batch Normalization，并去除了激活层而使用 Identity Mapping 或 Preactivation，进一步提升了网络性能。此后，Inception ResNet V2 融合了 Inception Net 优良的网络结构，和 ResNet 训练极深网络的残差学习模块，集两个方向之长，取得了更

好的分类效果。

我们可以看到，自 AlexNet 于 2012 年提出后，深度学习领域的研究发展极其迅速，基本上每年甚至每几个月都会出现新一代的技术。新的技术往往伴随着新的网络结构，更深的网络的训练方法等，并在图像识别等领域不断创造新的准确率记录。至今，ILSVRC 比赛和卷积神经网络的研究依然处于高速发展期，CNN 的技术日新月异。当然其中不可忽视的推动力是，我们拥有了更快的 GPU 计算资源用以实验，以及非常方便的开源工具（比如 TensorFlow）可以让研究人员快速地进行探索和尝试。在以前，研究人员如果没有像 Alex 那样高超的编程实力能自己实现 cuda-convnet，可能都没办法设计 CNN 或者快速地进行实验。现在有了 TensorFlow，研究人员和开发人员都可以简单而快速地设计神经网络结构并进行研究、测试、部署乃至实用。



7

TensorFlow 实现循环神经网络及 Word2Vec

本章我们将探索循环神经网络 (RNN) 和 Word2Vec⁵⁵，并在 TensorFlow 上实现它们。循环神经网络是在 NLP (Nature Language Processing, 自然语言处理) 领域最常使用的神经网络结构，和卷积神经网络在图像识别领域的地位类似。而 Word2Vec 则是将语言中的字词转化为计算机可以理解的稠密向量 (Dense Vector)，进而可以做其他自然语言处理任务，比如文本分类、词性标注、机器翻译等。

7.1 TensorFlow 实现 Word2Vec

Word2Vec 也称 Word Embeddings，中文有很多叫法，比较普便的是“词向量”或“词嵌入”。Word2Vec 是一个可以将语言中字词转为向量形式表达 (Vector Representations) 的模型，我们先来看看为什么要把字词转为向量。图像、音频等数据天然可以编码并存储为稠密向量的形式，比如图片是像素点的稠密矩阵，音频可以转为声音信号的频谱数据。自然语言处理在 Word2Vec 出现之前，通常将字词转成离散的单独的符号，比如将“中国”转为编号为 5178 的特征，将“北京”转为编号为 3987 的特征。这即是 One-Hot Encoder，

一个词对应一个向量（向量中只有一个值为 1，其余为 0），通常需要将一篇文章中每一个词都转成一个向量，而整篇文章则变为一个稀疏矩阵。对文本分类模型，我们使用 Bag of Words 模型，将文章对应的稀疏矩阵合并为一个向量，即把每一个词对应的向量加到一起，这样只统计每个词出现的次数，比如“中国”出现 23 次，那么第 5178 个特征为 23，“北京”出现 2 次，那么第 3987 个特征为 2。

使用 One-Hot Encoder 有一个问题，即我们对特征的编码往往是随机的，没有提供任何关联信息，没有考虑到字词间可能存在的关系。例如，我们对“中国”和“北京”的从属关系、地理位置关系等一无所知，我们从 5178 和 3987 这两个值看不出任何信息。同时，将字词存储为稀疏向量的话，我们通常需要更多的数据来训练，因为稀疏数据训练的效率比较低，计算也非常麻烦。使用向量表达（Vector Representations）则可以有效地解决这个问题。向量空间模型（Vector Space Models）可以将字词转为连续值（相对于 One-Hot 编码的离散值）的向量表达，并且其中意思相近的词将被映射到向量空间中相近的位置。向量空间模型在 NLP 中主要依赖的假设是 Distributional Hypothesis，即在相同语境中出现的词其语义也相近。向量空间模型可以大致分为两类，一类是计数模型，比如 Latent Semantic Analysis；另一类是预测模型（比如 Neural Probabilistic Language Models）。计数模型统计在语料库中，相邻出现的词的频率，再把这些计数统计结果转为小而稠密的矩阵；而预测模型则根据一个词周围相邻的词推测出这个词，以及它的空间向量。

Word2Vec 即是一种计算非常高效的，可以从原始语料中学习字词空间向量的预测模型。它主要分为 CBOW（Continuous Bag of Words）和 Skip-Gram 两种模式，其中 CBOW 是从原始语句（比如：中国的首都是____）推测目标字词（比如：北京）；而 Skip-Gram 则正好相反，它是从目标字词推测出原始语句，其中 CBOW 对小型数据比较合适，而 Skip-Gram 在大型语料中表现得更好。

使用 Word2Vec 训练语料能得到一些非常有趣的结果，比如意思相近的词在向量空间中的位置会接近。从一份 Google 训练超大语料得到的结果中看，诸如 Beijing、London、New York 等城市的名字会在向量空间中聚集在一起，而 Cat、Dog、Fish 等动物词汇也会聚集在一起。同时，如图 7-1 所示，Word2Vec 还能学会一些高阶的语言概念，比如我们计算“man”到“woman”的向量（词汇都是向量空间中的点，可计算两点间的向量），会发现它和“king”到“queen”的向量非常相似，即模型学到了男人与女人的关系；同时，“walking”到“walked”的向量和“swimming”到“swam”的向量非常相似，模型学到了进行时与过去时的关系。

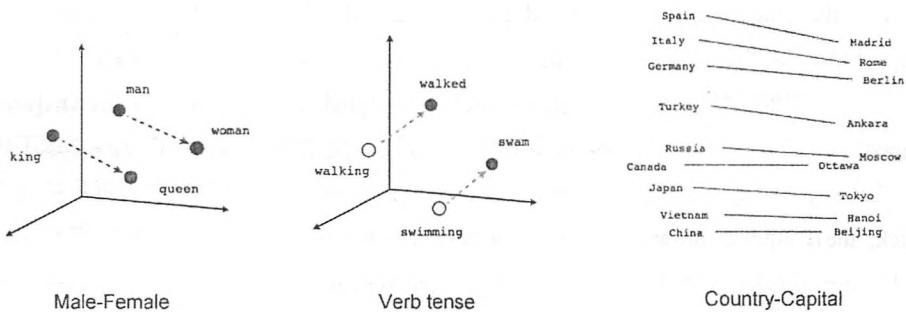


图 7-1 Word2Vec 模型可学习到的抽象概念

预测模型 Neural Probabilistic Language Models 通常使用最大似然的方法，在给定前面的语句 h 的情况下，最大化目标词汇 w_i 的概率。但它存在的一个比较严重的问题是计算量非常大，需要计算词汇表中所有单词出现的可能性。在 Word2Vec 的 CBOW 模型中，不需要计算完整的概率模型，只需要训练一个二元的分类模型，用来区分真实的目标词汇和编造的词汇（噪声）这两类，如图 7-2 所示。这种用少量噪声词汇来估计的方法，类似于蒙特卡洛模拟。

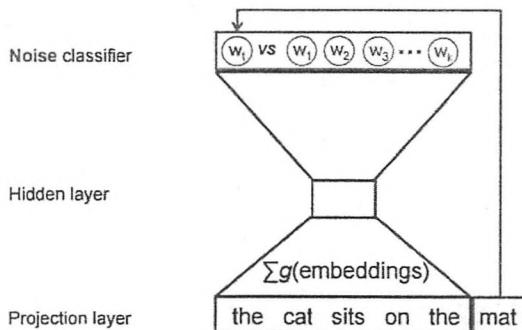


图 7-2 CBOW 模型结构示意图

当模型预测真实的目标词汇为高概率，同时预测其他噪声词汇为低概率时，我们训练的学习目标就被最优化了。用编造的噪声词汇训练的方法被称为 Negative Sampling。用这种方法计算 loss function 的效率非常高，我们只需要计算随机选择的 k 个词汇而非词汇表中的全部词汇，因此训练速度非常快。在实际中，我们使用 Noise-Contrastive Estimation (NCE) Loss，同时在 TensorFlow 中也有 `tf.nn.nce_loss()` 直接实现了这个 loss。

在本节中我们将主要使用 Skip-Gram 模式的 Word2Vec，先来看一下它训练样本的构

造，以“the quick brown fox jumped over the lazy dog”这句话为例。我们要构造一个语境与目标词汇的映射关系，其中语境包括一个单词左边和右边的词汇，假设我们的滑窗尺寸为1，可以制造的映射关系包括[the, brown] → quick、[quick, fox] → brown、[brown, jumped] → fox 等。因为 Skip-Gram 模型是从目标词汇预测语境，所以训练样本不再是[the, brown] → quick，而是 quick → the 和 quick → brown。我们的数据集就变为了(quick, the)、(quick, brown)、(brown, quick)、(brown, fox)等。我们训练时，希望模型能从目标词汇 quick 预测出语境 the，同时也需要制造随机的词汇作为负样本（噪声），我们希望预测的概率分布在正样本 the 上尽可能大，而在随机产生的负样本上尽可能小。这里的做法就是通过优化算法比如 SGD 来更新模型中 Word Embedding 的参数，让概率分布的损失函数（NCE Loss）尽可能小。这样每个单词的 Embedded Vector 就会随着训练过程不断调整，直到处于一个最适合语料的空间位置。这样我们的损失函数最小，最符合语料，同时预测出正确单词的概率也最高。

下面开始用 TensorFlow 实现 Word2Vec 的训练。首先依然是载入各种依赖库，这里因为要从网络下载数据，因此需要的依赖库比较多。本节代码主要来自 TensorFlow 的开源实现⁵⁶。

```
import collections
import math
import os
import random
import zipfile
import numpy as np
import urllib
import tensorflow as tf
```

我们先定义下载文本数据的函数。这里使用 `urllib.request.urlretrieve` 下载数据的压缩文件并核对文件尺寸，如果已经下载了文件则跳过。

```
url = 'http://mattmahoney.net/dc/'

def maybe_download(filename, expected_bytes):
    if not os.path.exists(filename):
        filename, _ = urllib.request.urlretrieve(url + filename, filename)
        statinfo = os.stat(filename)
```

```

if statinfo.st_size == expected_bytes:
    print('Found and verified', filename)
else:
    print(statinfo.st_size)
    raise Exception(
        'Failed to verify ' + filename + '. Can you get to it with a browser?')
return filename

filename = maybe_download('text8.zip', 31344016)

```

接下来解压下载的压缩文件，并使用 `tf.compat.as_str` 将数据转成单词的列表。通过程序输出，可以知道数据最后被转为了一个包含 17005207 个单词的列表。

```

def read_data(filename):
    with zipfile.ZipFile(filename) as f:
        data = tf.compat.as_str(f.read(f.namelist()[0])).split()
    return data

words = read_data(filename)
print('Data size', len(words))

```

接下来创建 `vocabulary` 词汇表，我们使用 `collections.Counter` 统计单词列表中单词的频数，然后使用 `most_common` 方法取 top 50000 频数的单词作为 `vocabulary`。再创建一个 `dict`，将 top 50000 词汇的 `vocabulary` 放入 `dictionary` 中，以便快速查询，Python 中 `dict` 查询复杂度为 O(1)，性能非常好。接下来将全部单词转为编号（以频数排序的编号），top 50000 词汇之外的单词，我们认定其为 `Unkown`（未知），将其编号为 0，并统计这类词汇的数量。下面遍历单词列表，对其中每一个单词，先判断是否出现在 `dictionary` 中，如果是则转为其编号，如果不是则转为编号 0（`Unkown`）。最后返回转换后的编码（`data`）、每个单词的频数统计（`count`）、词汇表（`dictionary`）及其反转的形式（`reverse_dictionary`）。

```

vocabulary_size = 50000

def build_dataset(words):
    count = [['UNK', -1]]
    count.extend(collections.Counter(words).most_common(vocabulary_size - 1))

```

```

dictionary = dict()
for word, _ in count:
    dictionary[word] = len(dictionary)
data = list()
unk_count = 0
for word in words:
    if word in dictionary:
        index = dictionary[word]
    else:
        index = 0
        unk_count += 1
    data.append(index)
count[0][1] = unk_count
reverse_dictionary = dict(zip(dictionary.values(), dictionary.keys()))
return data, count, dictionary, reverse_dictionary
data, count, dictionary, reverse_dictionary = build_dataset(words)

```

然后我们删除原始单词列表，可以节约内存。再打印 vocabulary 中最高频出现的词汇及其数量（包括 Unknown 词汇），可以看到“UNK”这类一共有 418391 个，最常出现的“the”有 1061396 个，排名第二的“of”有 593677 个。我们的 data 中前 10 个单词为['anarchism', 'originated', 'as', 'a', 'term', 'of', 'abuse', 'first', 'used', 'against']，对应的编号为[5235, 3084, 12, 6, 195, 2, 3137, 46, 59, 156]

```

del words
print('Most common words (+UNK)', count[:5])
print('Sample data', data[:10], [reverse_dictionary[i] for i in data[:10]])

```

下面生成 Word2Vec 的训练样本。我们根据前面提到的 Skip-Gram 模式（从目标单词反推语境），将原始数据“the quick brown fox jumped over the lazy dog”转为(quick, the)、(quick, brown)、(brown, quick)、(brown, fox)等样本。我们定义函数 generate_batch 用来生成训练用的 batch 数据，参数中 batch_size 为 batch 的大小；skip_window 指单词最远可以联系的距离，设为 1 代表只能跟紧邻的两个单词生成样本，比如 quick 只能和前后的单词生成两个样本(quick,the)和(quick,brown); num_skips 为对每个单词生成多少个样本，它不能大于 skip_window 值的两倍，并且 batch_size 必须是它的整数倍（确保每个 batch

包含了一个词汇对应的所有样本)。我们定义单词序号 `data_index` 为 `global` 变量, 因为我们反复调用 `generate_batch`, 所以要确保 `data_index` 可以在函数 `generate_batch` 中被修改。我们也使用 `assert` 确保 `num_skips` 和 `batch_size` 满足前面提到的条件。然后用 `np.ndarray` 将 `batch` 和 `labels` 初始化为数组。这里定义 `span` 为对某个单词创建相关样本时会使用到的单词数量, 包括目标单词本身和它前后的单词, 因此 `span=2*skip_window+1`。并创建一个最大容量为 `span` 的 `deque`, 即双向队列, 在对 `deque` 使用 `append` 方法添加变量时, 只会保留最后插入的 `span` 个变量。

```
data_index = 0

def generate_batch(batch_size, num_skips, skip_window):
    global data_index
    assert batch_size % num_skips == 0
    assert num_skips <= 2 * skip_window
    batch = np.ndarray(shape=(batch_size), dtype=np.int32)
    labels = np.ndarray(shape=(batch_size, 1), dtype=np.int32)
    span = 2 * skip_window + 1
    buffer = collections.deque(maxlen=span)
```

接下来从序号 `data_index` 开始, 把 `span` 个单词顺序读入 `buffer` 作为初始值。因为 `buffer` 是容量为 `span` 的 `deque`, 所以此时 `buffer` 已填充满, 后续数据将替换掉前面的数据。然后我们进入第一层循环(次数为 `batch_size//num_skips`), 每次循环内对一个目标单词生成样本。现在 `buffer` 中是目标单词和所有相关单词, 我们定义 `target=skip_window`, 即 `buffer` 中第 `skip_window` 个变量为目标单词。然后定义生成样本时需要避免的单词列表 `targets_to_avoid`, 这个列表一开始包括第 `skip_window` 个单词(即目标单词), 因为我们预测的是语境单词, 不包括目标单词本身。接下来进入第二层循环(次数为 `num_skips`), 每次循环中对一个语境单词生成样本, 先产生随机数, 直到随机数不在 `targets_to_avoid` 中, 代表可以使用的语境单词, 然后产生一个样本, `feature` 即目标词汇 `buffer[skip_window]`, `label` 则是 `buffer[target]`。同时, 因为这个语境单词被使用了, 所以再把它添加到 `targets_to_avoid` 中过滤。在对一个目标单词生成完所有样本后(`num_skips` 个样本), 我们再读入下一个单词(同时会抛掉 `buffer` 中第一个单词), 即把滑窗向后移动一位, 这样我们的目标单词也向后移动了一个, 语境单词也整体后移了, 便可以开始生成下一个目标单词的训练样本。两层循环完成后, 我们已经获得了 `batch_size` 个训练样本, 将 `batch` 和 `labels`

作为函数结果返回。

```

for _ in range(span):
    buffer.append(data[data_index])
    data_index = (data_index + 1) % len(data)
for i in range(batch_size // num_skips):
    target = skip_window
    targets_to_avoid = [skip_window]
    for j in range(num_skips):
        while target in targets_to_avoid:
            target = random.randint(0, span - 1)
        targets_to_avoid.append(target)
        batch[i * num_skips + j] = buffer[skip_window]
        labels[i * num_skips + j, 0] = buffer[target]
    buffer.append(data[data_index])
    data_index = (data_index + 1) % len(data)
return batch, labels

```

这里调用 generate_batch 函数简单测试一下其功能。参数中将 batch_size 设为 8，num_skips 设为 2，skip_window 设为 1，然后执行 generate_batch 并获得 batch 和 labels。再打印 batch 和 labels 的数据，可以看到我们生成的样本是“3084 originated -> 5235 anarchism”，“3084 originated -> 12 as”，“12 as -> 3084 originated”等。以第一个样本为例，3084 是目标单词 originated 的编号，这个单词对应的语境单词是 anarchism，其编号为 5235。

```

batch, labels = generate_batch(batch_size=8, num_skips=2, skip_window=1)
for i in range(8):
    print(batch[i], reverse_dictionary[batch[i]], '->', labels[i, 0],
          reverse_dictionary[labels[i, 0]])

```

我们定义训练时的 batch_size 为 128；embedding_size 为 128，embedding_size 即将单词转为稠密向量的维度，一般是 50~1000 这个范围内的值，这里使用 128 作为词向量的维度；skip_window 即前面提到的单词间最远可以联系的距离，设为 1；num_skips 即对每个目标单词提取的样本数，设为 2。然后我们再生成验证数据 valid_examples，这里随机抽取一些频数最高的单词，看向量空间上跟它们最近的单词是否相关性比较高。

`valid_size=16` 指用来抽取的验证单词数, `valid_window=100` 是指验证单词只从频数最高的 100 个单词中抽取, 我们使用 `np.random.choice` 函数进行随机抽取。而 `num_sampled` 是训练时用来做负样本的噪声单词的数量。

```
batch_size = 128
embedding_size = 128
skip_window = 1
num_skips = 2

valid_size = 16
valid_window = 100
valid_examples = np.random.choice(valid_window, valid_size, replace=False)
num_sampled = 64
```

下面就开始定义 Skip-Gram Word2Vec 模型的网络结构。我们先创建一个 `tf.Graph` 并设置为默认的 `graph`。然后创建训练数据中 `inputs` 和 `labels` 的 `placeholder`, 同时将前面随机产生的 `valid_examples` 转为 TensorFlow 中的 `constant`。接下来, 先使用 `with tf.device('/cpu:0')` 限定所有计算在 CPU 上执行, 因为接下去的一些计算操作在 GPU 上可能还没有实现。然后使用 `tf.random_uniform` 随机生成所有单词的词向量 `embeddings`, 单词表大小为 50000, 向量维度为 128, 再使用 `tf.nn.embedding_lookup` 查找输入 `train_inputs` 对应的向量 `embed`。下面使用之前提到的 NCE Loss 作为训练的优化目标, 我们使用 `tf.truncated_normal` 初始化 NCE Loss 中的权重参数 `nce_weights`, 并将其 `nce_biases` 初始化为 0。最后使用 `tf.nn.nce_loss` 计算学习出的词向量 `embedding` 在训练数据上的 loss, 并使用 `tf.reduce_mean` 进行汇总。

```
graph = tf.Graph()
with graph.as_default():

    train_inputs = tf.placeholder(tf.int32, shape=[batch_size])
    train_labels = tf.placeholder(tf.int32, shape=[batch_size, 1])
    valid_dataset = tf.constant(valid_examples, dtype=tf.int32)

    with tf.device('/cpu:0'):
        embeddings = tf.Variable(
```

```

    tf.random_uniform([vocabulary_size, embedding_size], -1.0, 1.0))
embed = tf.nn.embedding_lookup(embeddings, train_inputs)

nce_weights = tf.Variable(
    tf.truncated_normal([vocabulary_size, embedding_size],
                        stddev=1.0 / math.sqrt(embedding_size)))
nce_biases = tf.Variable(tf.zeros([vocabulary_size]))

loss = tf.reduce_mean(tf.nn.nce_loss(weights=nce_weights,
                                      biases=nce_biases,
                                      labels=train_labels,
                                      inputs=embed,
                                      num_sampled=num_sampled,
                                      num_classes=vocabulary_size))

```

我们定义优化器为 SGD，且学习速率为 1.0。然后计算嵌入向量 embeddings 的 L2 范数 norm，再将 embeddings 除以其 L2 范数得到标准化后的 normalized_embeddings。再使用 tf.nn.embedding_lookup 查询验证单词的嵌入向量，并计算验证单词的嵌入向量与词汇表中所有单词的相似性。最后，我们使用 tf.global_variables_initializer 初始化所有模型参数。

```

optimizer = tf.train.GradientDescentOptimizer(1.0).minimize(loss)

norm = tf.sqrt(tf.reduce_sum(tf.square(embeddings), 1, keep_dims=True))
normalized_embeddings = embeddings / norm
valid_embeddings = tf.nn.embedding_lookup(
    normalized_embeddings, valid_dataset)
similarity = tf.matmul(
    valid_embeddings, normalized_embeddings, transpose_b=True)

init = tf.global_variables_initializer()

```

我们定义最大的迭代次数为 10 万次，然后创建并设置默认的 session，并执行参数初始化。在每一步训练迭代中，先使用 generate_batch 生成一个 batch 的 inputs 和 labels 数据，

并用它们创建 feed_dict。然后使用 session.run() 执行一次优化器运算（即一次参数更新）和损失计算，并将这一步训练的 loss 累积到 average_loss。

```
num_steps = 100001

with tf.Session(graph=graph) as session:
    init.run()
    print("Initialized")

    average_loss = 0
    for step in range(num_steps):
        batch_inputs, batch_labels = generate_batch(
            batch_size, num_skips, skip_window)
        feed_dict = {train_inputs : batch_inputs, train_labels : batch_labels}

        _, loss_val = session.run([optimizer, loss], feed_dict=feed_dict)
        average_loss += loss_val
```

之后每 2000 次循环，计算一下平均 loss 并显示出来。

```
if step % 2000 == 0:
    if step > 0:
        average_loss /= 2000
    print("Average loss at step ", step, ":", average_loss)
    average_loss = 0
```

每 10000 次循环，计算一次验证单词与全部单词的相似度，并将与每个验证单词最相似的 8 个单词展示出来。

```
if step % 10000 == 0:
    sim = similarity.eval()
    for i in range(valid_size):
        valid_word = reverse_dictionary[valid_examples[i]]
        top_k = 8
        nearest = (-sim[i, :]).argsort()[1:top_k+1]
        log_str = "Nearest to %s:" % valid_word
```

```

for k in range(top_k):
    close_word = reverse_dictionary[nearest[k]]
    log_str = "%s %s," % (log_str, close_word)
    print(log_str)
final_embeddings = normalized_embeddings.eval()

```

以下为展示出来的平均损失，以及与验证单词相似度最高的单词，可以看到我们训练的模型对名词、动词、形容词等类型的单词的相似词汇的识别都非常准确。因此由 Skip-Gram Word2Vec 得到的向量空间表达（Vector Representations）是非常高质量的，近义词在向量空间上的位置也是非常靠近的。

```

Average loss at step 92000 : 4.70622572589
Average loss at step 94000 : 4.61680726242
Average loss at step 96000 : 4.73945830989
Average loss at step 98000 : 4.63924189049
Average loss at step 100000 : 4.67957950294
Nearest to five: six, four, seven, eight, three, zero, two, nine,
Nearest to state: government, amalthea, habsburg, asparagales, cegep, barrac
uda, dasyprocta, connecticut,
Nearest to over: three, reginae, from, replace, trapezohedron, around, brine,
    it,
Nearest to were: are, have, had, was, while, been, be, wct,
Nearest to at: in, on, mitral, agouti, triglycerides, excerpts, during, with
    in,
Nearest to called: agouti, akita, homeworld, layouts, dasyprocta, UNK, cegep,
    referred,
Nearest to about: disclosed, antimatter, vec, advocated, surgeries, defiance,
    disband, legionnaire,
Nearest to which: that, this, gollancz, but, what, also, it, and,
Nearest to three: four, five, six, two, seven, eight, iit, nine,
Nearest to that: which, however, what, this, when, gollancz, but, ramps,
Nearest to new: nonviolent, aquila, assyrian, gardening, local, charcot, sub
    sistence, ssbn,
Nearest to eight: seven, six, nine, four, five, zero, three, mitral,

```