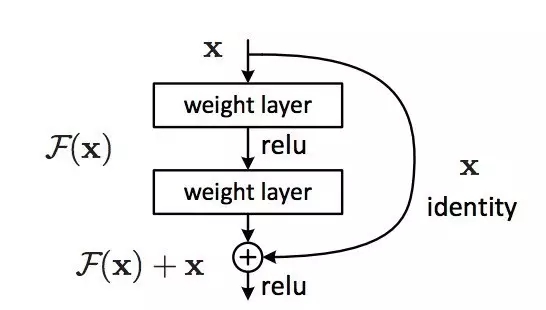
**TensorFlow实现ResNet深度残差网络**

1. 背景介绍

ResNet(Residual Neural Network)是由微软研究院的何凯明博士等4名华人提出，通过引入了残差学习单元，成功训练了152层极深的深度神经网络。ResNet还获得了2015年的ImageNet上的分类，检测，定位任务的冠军。从这时开始，神经网络可以向极深的方向发展，甚至能达到1000多层。后来的GoogLeNet Inception V4中也借鉴了这一结构。可以说ResNet的出现是神经网络的一个里程碑的突破。

1. 相关技术简介
2. 什么是残差学习单元?

对于一个很深的神经网络而言，随着网络的不断加深，会出现一个退化的问题，即随着网络层数增加，准确率会不断趋近于饱和，再增加深度时就会使得神经网络在训练集上的正确率大幅度下降，为了解决这个问题，ResNet的作者何凯明博士引入了残差学习单元的概念，如下图所示，作者的思路是加入恒等映射(identity mapping)，恒等映射即令输出等于输入，为什么要加入恒等映射呢?假设我们现在有一个性能还不错的神经网络，这时我们在该神经网络的后面加入一些用来构造恒等映射的非线性层，即使可能中间某些层没有发挥太大的作用，但是这个加深的神经网络的性能一定是大于等于浅层神经网络的，VGGNet就证明了这一点，那么问题来了，神经网络的加深势必导致训练难度的增加，而且这个困难是指数式增长，那么我们该如何解决这个问题呢?作者的思路是这样的，把原先想要拟合的函数H(x)转换为F(x)+x，那么F(x) = H(x)-x，其中x代表原始输入，F(x)代表残差函数，这样，中间的非线性层由之前拟合H(x)变为拟合F(x)，最终目的是将F(x)逼近于0，这样就构造成了恒等映射，尽管最后拟合的都近似H(x)，但是学习的复杂度是存在着很大不同的，打个比方，原先我们所拟合的映射从H(100) = 101 变为 H(100) = 102，这个变化率才有1/101，但是引入残差之后，拟合的映射从F(100) = 1 变为 F(100) = 2，变化率有100%，可以看出引入残差之后训练的复杂程度大大减小。

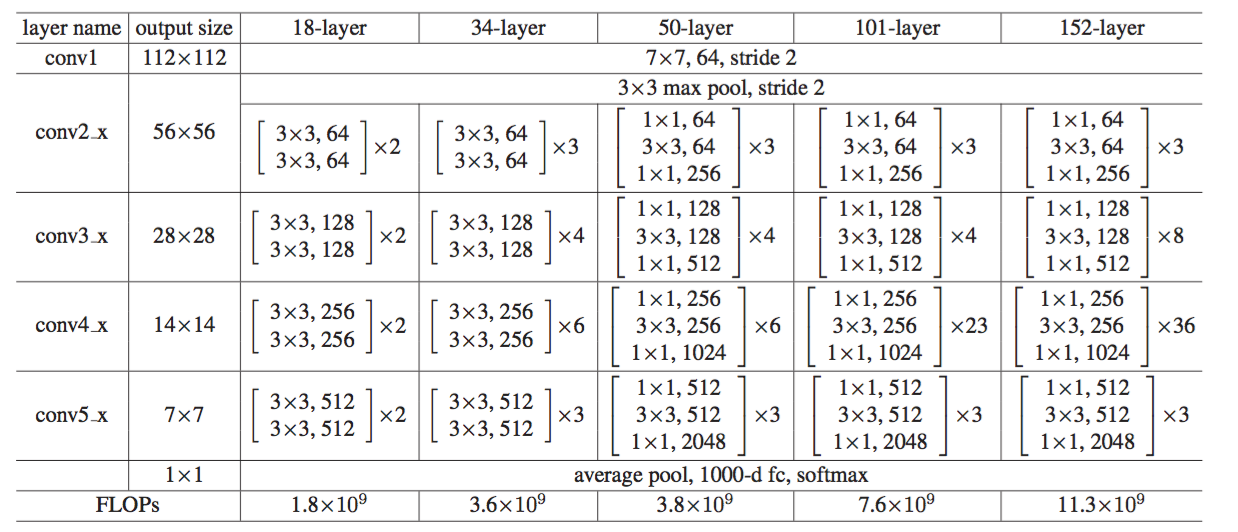


接下来让我们来看一下深度残差网络与传统神经网络的一些不同。我们可以看到，在残差网络中会出现一些旁路直接连到后面的层，这种直连通道的结构相当于是shortcut，整个网络只需要学习输入与输出的残差。这样大大简化了学习的复杂度。注意的是，这样的残差学习单元应保持输入和输出的维度一致。



1. ResNet的组成结构是怎样的?

由上图我们已经大致了解了ResNet的组成，即由很多残差学习单元堆叠而成的神经网络，从下图中我们可以看到，ResNet基本是由两种残差学习单元组成，一种是3\*3卷积的二层残差学习单元，另一种是由1\*1，3\*3，1\*1所连接的三层学习单元，其中三层学习单元的1\*1卷积核主要作用是先降维再升维，使得最终的输入，输出维度相匹配。



接下来让我们实现ResNet V2，V2与V1的最大区别是V2在每层中都使用了批标准化，这样可以得残差学习单元更容易训练并且防止过拟合，同时，V2把shortcut中的非线性激活函数被替换为恒等映射(y=x)，这是因为前馈和反馈信号可以直接传输，可以节省非线性计算的复杂度。

3. 实验目标

利用TensorFlow平台高级组件contrib.slim库来实现ResNet V2神经网络，并测试神经网络的前馈与反馈的时间。

1. 实验步骤

本次实验我们分为三个步骤:

1) 载入所需要的库并预定义操作

2) 构建神经网络

3) 评价神经网络前馈与反馈的训练时间

(本节代码主要来源于TensorFlow的开源实现)

1) 载入所需要的库并预定义操作

本次实验涉及到的操作比较多，因此使用contrib.slim库来方便地创建ResNet。首先让我们导入本次实验所需要的库。其中collections库是python原生库，能方便地创建一个集合。

|  |
| --- |
| import collections  from datetime import datetime  import math  import time  import tensorflow as tf  slim = tf.contrib.slim |

接下来让我们定义一个残差学习模块类，其中collections.namedtuple函数代表创建一命名为Block的元组，其中含有scope(残差学习单元的名称),，unit\_fn(类型为残差学习单元)，args(该残差学习单元的参数)这几个命名元素，其中args列表中的每个元素都代表一个残差学习单元。以Block(*'block1'*, bottleneck, [(256,64,1)]\*2 + [(256,64,2)])为例，我们创建了一个名为’block1’的残差学习模块，其中含有三个学习单元，每个残差学习单元都包含三个卷积层，以(256,64,3)为例，256代表第三层输出为256,64代表前两层输出为64,3代表中间那层的步长为3。这个构建函数我们会在后面具体看到。

|  |
| --- |
| class **Block**(collections.namedtuple(*'Block'*, [*'scope'*, *'unit\_fn'*, *'args'*])):  *'''ResNet block'''* |

接下来我们定义一个函数来使用池化层来降采样，inputs代表输入，factor代表将输出的tensor降为1/factor，factor=1即不用降采样，这样可以防止过拟合。

|  |
| --- |
| def **subsample**(inputs, factor, scope=None):  if factor == 1:  return inputs  else:  return slim.max\_pool2d(inputs, [1,1], stride=factor, scope=scope) |

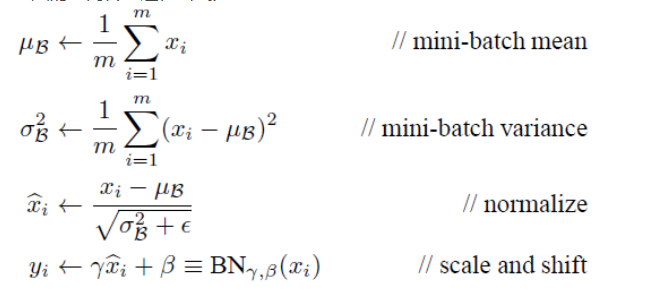
接下来我们定义conv2d\_same函数来创建卷积层，这几个参数相信大家已经很眼熟了，这里就不一一介绍了。接下来定义如果卷积核步长为1，则直接使用SAME方式进行卷积，如果步长不为1，那么在卷积时很可能出现剩下的区域不够卷积的问题，这里我们使用tf.pad函数进行补0操作，其中inputs代表输入，后面的列表中的四个列表分别代表Tensor的四个输入维度(batch\_size, 宽，高，通道数量)，接下来的每个对应维度的列表[x, y]向该维度正方向填充x个维度，负方向填充y个维度，以inputs = tf.pad(inputs, [[0,0], [pad\_beg, pad\_end], [pad\_beg, pad\_end],[0,0]])为例，意思是分别向图片上方填充pad\_beg行，下方填充pad\_end行，再向图片左面填充pad\_beg列，右面填充pad\_end列，这是为了让图片无法被卷积到的图片卷积到(我个人认为直接用SAME方式也可以的。。。)

|  |
| --- |
| def **conv2d\_same**(inputs, num\_outputs, kernel\_size, stride, scope=None):  if stride == 1:  return slim.conv2d(inputs, num\_outputs, kernel\_size, stride=1, 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) |

接下来我们定义一个函数用来堆叠残差学习单元模块组(Blocks)的函数stack\_blocks\_dense，其中net是输入，blocks是我们所定义的残差学习单元模块组，outputs\_collections是用来收集各个模块输出层的列表。接下来遍历模块组中所有的模块中的所有残差学习单元中的配置，并使用unit\_fn函数顺序创建并连接所有的残差学习单元，注意这里unit\_fn并不是自带的函数，他只是一个命名为unit\_fn的后面的bottleneck函数，最终使用collect\_named\_outputs函数将这些残差学习单元模块添加到集合中，并返回最终的网络net，同时不要忘记在函数前加上@slim.add\_arg\_scope 装饰器，该修饰器的作用是方便使用slim.arg\_scope 函数直接为函数传入参数，这个在后面会用到。

|  |
| --- |
| *@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(tuple(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 |

接下来我们创建函数resnet\_arg\_scope用来定义ResNet网络的默认值，这样我们可以在后面节省相当多的代码，同时在这里我们使用到了BN算法，该算法认为随着网络权重的不断更新，每层的输出分布均会发生改变，最终可能因测试集上数据分布与原数据分布不同而导致模型的泛化能力变弱，同时若训练数据集的每个batch块数据分布不同，那么可能使得神经网络的迭代次数增加，使得神经网络无法收敛。该算法在每层的输入之前先进行预标准化以使得每层的输入分布情况相同，即使得每一维的均值为0，标准差为1。整个过程近似如下:



接下来让我们回到代码，其中参数is\_training=True代表设置训练标记为True，batch\_norm\_decay代表批标准化中的的权重衰减速率，这部分起到了惩罚权重的作用，即控制权重不至于太大同时使得作用不大的权重逐渐变为0，batch\_norm\_epsilon是批标准化中设置的一个常量，batch\_norm\_scale=True允许平移和缩放(过程很复杂，有兴趣可以自行研究下BN算法)，接下来设置标准化器的参数字典batch\_norm\_params，字典中所含有的变量的值均由函数参数给出。接下来我们定义卷积层，默认参数为使用L2正则化(L2正则化也是一个比较复杂的数学过程，可以自行查阅)L2正则化只影响权重的更新而不影响偏置的更新。主要目的是来约束权重的范围，这个作用跟BN算法中的权重衰减是一样的，都是用来防止过拟合，接下来使用slim.arg\_scope函数来共享参数，该函数的第一个参数时是一个列表，列表里面是所需要共享参数的函数，其余参数为该函数默认的参数, 也就是说在该作用域下定义的列表中的函数都会默认使用这些参数值。接下来我们将卷积层的权重的初始化器选择slim.variance\_scaling\_initializer()，激活函数设置为ReLU，标准化器为BN(Batch\_Normalization)，标准化器的参数为batch\_norm\_params，BN层的参数设置为我们定义好的参数，最大池化层边缘的处理方式设置为’SAME’，最后将这几个嵌套的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 |

接下来让我们定义生成残差学习单元的函数bottleneck，参数inputs代表输入，depth，depth\_bottleneck，stride分别代表之前的残差学习单元参数。Outputs\_collections用来存储残差学习单元，最后的scope代表这个残差学习单元的名称。接下来使用slim.utils.last\_dimension来获取最后一个维度，即输出通道数，min\_rank=4参数是使输入至少为4个维度，接下来进行BN，即批标准化，在这里直接使用slim.batch\_norm函数进行批标准化即可，其中activation\_fn函数为预激活函数，即把激活函数层放在批标准化层前面，这样做的效果比放在后面的效果要好很多。接下来我们定义shortcut直连通道，因为中间可能经过步长不为1的卷积操作而导致图片的尺寸发生改变，因此这里我们先判断残差学习单元的输入维度和输出维度是否一致，注意这里的输入输出跟图片的尺寸是没有关系的，输入输出只跟卷积核的数量(或是高级特征的数量有关)，一致的话根据步长使用subsample函数来进行降采样，如果不一致的话还要通过1\*1的卷积核操作来进行升未或降维操作，这样我们的shortcut就定义好了，接下来定义残差函数部分，根据之前所讲过的内容，残差部分即拟合H(x)-x，这里的残差一共有三层卷积，第一个卷积尺寸为1\*1，步长为1，输出通道数为depth\_bottleneck，第二层卷积尺寸为3\*3，步长为stride，输出通道数为depth\_bottleneck，第三层卷积为1\*1，步长为1，输出通道数为depth\_bottleneck，因前面定义了BN层，所以在这里我们无需使用正则项或是Dropout来防止过拟合。最终将残差与shortcut相加来拟合原先想要拟合的函数并添加到列表中。

|  |
| --- |
| *@slim.add\_arg\_scope*  #残差学习单元  def **bottleneck**(inputs, depth, depth\_bottleneck, stride, outputs\_collections=None, scope=None):  with tf.variable\_scope(scope, *'bottleneck\_v2'*, [inputs]) as sc:  #获取输入的最后一个维度数(输入通道数) 至少为4个维度  depth\_in = slim.utils.last\_dimension(inputs.get\_shape(), min\_rank=4)  #批标准化&使用激活函数进行预处理(预激活)  preact = slim.batch\_norm(inputs, activation\_fn=tf.nn.relu, scope=*'preact'*)  #如果残差学习单元输入通道数和输出通道数相同，进行降采样处理  if depth == depth\_in:  #shortcut 直连通道 根据stride步长降采样  shortcut = subsample(inputs, stride, *'shortcut'*)  else:  shortcut = slim.conv2d(preact, depth, [1,1], stride=stride, normalizer\_fn=None, activation\_fn=None, scope=*'shortcut'*)  #定义残差部分  residual = slim.conv2d(preact, depth\_bottleneck, [1,1], stride=1, scope=*'conv1'*)  residual = conv2d\_same(residual, depth\_bottleneck, 3, stride, scope=*'conv2d'*)  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) |

接下来我们定义生成ResNet的主函数，inputs代表输入，blocks代表我们所定义的残差学习单元模块列表，num\_classes代表最终分类数，global\_pool代表是否在最后使用全局平均池化，include\_root\_block标志是是否在之前加入7\*7，步长为2的卷积层及池化层，reuse标志是否可以复用变量，scope指的是整个网络的名称，接下来在该作用域内我们定义end\_points\_collections命名列表用以存放残差学习单元模块，便于堆叠残差学习单元。接下来开始创建ResNet网络，我们首先创建最前面的输出为64，卷积尺寸为7\*7，步长为2的卷积层，再连接一个步长为2的3\*3最大池化，连续经过了两次步长为2的层，使得图片尺寸变为原先的1/4，然后使用stack\_blocks\_dense函数来生成残差学习单元模块组，然后添加平均池化层，这里也可以使用tf.avg\_pool函数来池化，不过使用tf.reduce\_mean(这个函数是用来求均值的)效率会更高一点，下面添加一个输出维度为num\_classes分类数的1\*1卷积，再添加一个Softmax层，最终把end\_points\_collections转换为字典，再把Softmax层加入其中，最终返回Resnet和end\_points\_collections。

|  |
| --- |
| 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 |

接下来定义网络的配置，分别定义50层，101层，152层，200层，以resnet\_v2\_50为例，50层=1个7\*7卷积+1个3\*3池化+3\*(block1中3个卷积层+block2中4个卷积层+block3中6个卷积层+block4中3个卷积层)。

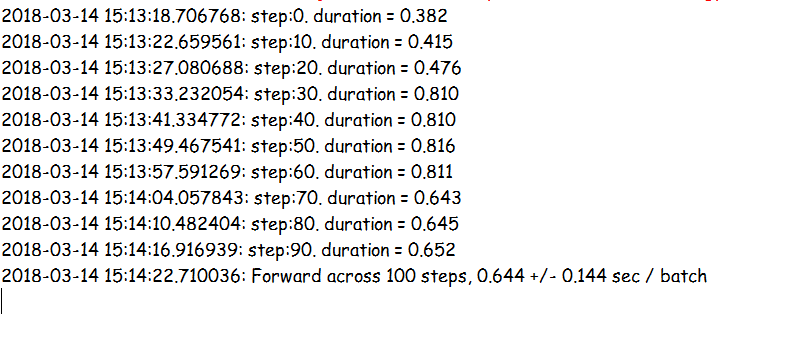
|  |
| --- |
| def **resnet\_v2\_50**(inputs, num\_classes=None, global\_pool=True, reuse=None, scope=*'resnet\_v2\_50'*):  blocks = [  Block(*'block1'*, bottleneck, [(256,64,1)]\*2 + [(256,64,2)]),  Block(*'block2'*, bottleneck, [(512, 128, 1)]\*3 + [(512, 128, 2)]),  Block(*'block3'*, bottleneck, [(1024, 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)  def **resnet\_v2\_101**(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)]\*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)  def **resnet\_v2\_152**(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)]\*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)  def **resnet\_v2\_200**(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)]\*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) |

接下来进行网络的评测，与前面卷积神经网络的评测方式相同。

|  |
| --- |
| def **time\_tensorflow\_run**(session, target, info\_string):  #预热轮数 增加cache  num\_steps\_burn\_in = 10  #总时间  total\_duration = 0.0  #计算方差  total\_duration\_squared = 0.0    for i in range(num\_steps\_burn\_in + num\_batches):  start\_time = time.time()  \_ = session.run(target)  duration = time.time() - start\_time  if i>= num\_steps\_burn\_in:  if not i % 10:  print(*r'%s: step:%d. duration = %.3f'* % (datetime.now(), i - num\_steps\_burn\_in, duration))  total\_duration += duration  total\_duration\_squared += duration \* duration    #每轮迭代的平均耗时  mn = total\_duration / num\_batches  #标准差  vr = total\_duration\_squared / num\_batches - mn\*mn  sd = math.sqrt(vr)  print(*r'%s: %s across %d steps, %.3f +/- %.3f sec / batch'* % (datetime.now(), info\_string, num\_batches, mn, sd))  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)  with tf.Session() as sess:  tf.global\_variables\_initializer().run()  num\_batches=100  time\_tensorflow\_run(sess, net, *'Forward'*) |

1. 实验结论

实验结果如下:



我们可以看到ResNet虽然有152层深，但是在forward训练的过程中并没有很复杂的计算，从ResNet开始，神经网络开始正式支持超神神经网络的训练，有趣的是，有一篇论文指出，ResNet并不是表面上很深的神经网络，该神经网络的关键路径很短，可以看做是一些浅层网络的集合的投票机制，但是不论怎么说，ResNet已经成为了深度学习发展的一个里程碑，在今天仍有可借鉴的地方。