

## CS231n

- numpy操作
- numpy的广播机制
- 线性分类
- 对图像均值化
- 激活函数
- 权重初始化
- 图片 可视化
- jupyter notebook中重新加载
- 训练集和验证集准确率
- 特征可视化
- 参数更新
- 学习率退火
- 调参
- 迁移学习
- 网络架构
- CNN中感受视野的计算
- Dilated Residual Networks
- 反卷积(Deconvolution)上采样(Upsampling)上池化(Unpooling)的区别
- 学习率设置

# CS231n

## numpy操作

1. np.sum()
  - 默认axis=None是所有元素进行求和
2. 取矩阵的某一列
  - $a[:, 1]$
3. numpy.random.choice(a, size=None, replace=True, p=None)  
-从一维数组 a 中选出 size 个元素，a中每个元素被选中的概率由一维数组p定义,replace=True: 可以从a 中反复选取同一个元素。  
replace=False: a 中同一个元素只能被选取一次。

## numpy的广播机制

1. 一般广播规则

- 在两个数组上运行时，NumPy将元素的形状进行比较。它从尾随的维度开始，并向前推进。两个尺寸兼容
    - 他们是平等的
    - 其中一个是1
- 执行 broadcast 的前提在于，两个 ndarray 执行的是 element-wise (按位加，按位减) 的运算，而不是矩阵乘法的运算，矩阵乘法运算时需要维度之间严格匹配
  - 如果上述规则产生有效结果，并且满足以下条件之一，那么数组被称为可广播的。
    - 数组拥有相同形状。
    - 数组拥有相同的维数，每个维度拥有相同长度，或者长度为 1。
    - 数组拥有极少的维度，可以在其前面追加长度为 1 的维度，使上述条件成立。
- 

## 线性分类

### 1. Softmax分类器

$f(x_i; W) = Wx_i$  保持不变，但将这些评分值视为每个分类的未归一化的对数概率，并且将折叶损失 (hinge loss) 替换为交叉熵损失 (cross-entropy loss)。公式如下：

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right) \text{ 或等价的 } L_i = -f_{y_i} + \log\left(\sum_j e^{f_j}\right)$$

在上式中，使用  $f_j$  来表示分类评分向量  $f$  中的第  $j$  个元素。和之前一样，整个数据集的损失值是数据集中所有样本数据的损失值  $L_i$  的均值与正则化损失  $R(W)$  之和。其中函数  $f_j(z) = \frac{e^{z_j}}{\sum_k e^{z_k}}$  被称作 softmax 函数：其输入值是一个向量，向量中元素为任意实数的评分值 ( $z$  中的)，函数对其进行压缩，输出一个向量，其中每个元素值在 0 到 1 之间，且所有元素之和为 1。所以，包含 softmax 函数的完整交叉熵损失看起来唬人，实际上还是比较容易理解的。

概率论解释：先看下面的公式：

$$P(y_i|x_i, W) = \frac{e^{f_{y_i}}}{\sum_j e^{f_j}}$$

可以解释为是给定图像数据  $x_i$ ，以  $W$  为参数，分配给正确分类标签  $y_i$  的归一化概率。为了理解这点，请回忆一下 Softmax 分类器将输出向量  $f$  中的评分值解释为没有归一化的对数概率。那么以这些数值做指数函数的幂就得到了没有归一化的概率，而除法操作则对数据进行了归一化处理，使得这些概率的和为 1。从概率论的角度来理解，我们就是在最小化正确分类的负对数概率，这可以看做是在进行最大似然估计 (MLE)。该解释的另一个好处是，损失函数中的正则化部分  $R(W)$  可以被看做是权重矩阵  $W$  的高斯先验，这里进行的是最大后验估计 (MAP) 而不是最大似然估计。提及这些解释只是为了让读者形成直观的印象，具体细节就超过本课程范围了。

**实操事项：数值稳定。**编程实现softmax函数计算的时候，中间项  $e^{f_{y_i}}$  和  $\sum_j e^{f_j}$  因为存在指数函数，所以数值可能非常大。除以大数值可能导致数值计算的不稳定，所以学会使用归一化技巧非常重要。如果在分式的分子和分母都乘以一个常数  $C$ ，并把它变换到求和之中，就能得到一个从数学上等价的公式：

$$\frac{e^{f_{y_i}}}{\sum_j e^{f_j}} = \frac{Ce^{f_{y_i}}}{C \sum_j e^{f_j}} = \frac{e^{f_{y_i} + \log C}}{\sum_j e^{f_j + \log C}}$$

$C$  的值可自由选择，不会影响计算结果，通过使用这个技巧可以提高计算中的数值稳定性。通常将  $C$  设为  $\log C = -\max_j f_j$ 。该技巧简单地说，就是应该将向量  $f$  中的数值进行平移，使得最大值为0。代码实现如下：

```
f = np.array([123, 456, 789]) # 例子中有3个分类，每个评分的数值都很大
p = np.exp(f) / np.sum(np.exp(f)) # 不妙：数值问题，可能导致数值爆炸

# 那么将f中的值平移到最大值为0：
f -= np.max(f) # f becomes [-666, -333, 0]
p = np.exp(f) / np.sum(np.exp(f)) # 现在OK了，将给出正确结果
```

## 2. SVM分类器

让我们更精确一些。回忆一下，第*i*个数据中包含图像  $x_i$  的像素和代表正确类别的标签  $y_i$ 。评分函数输入像素数据，然后通过公式  $f(x_i, W)$  来计算不同分类类别的分值。这里我们将分值简写为  $s$ 。比如，针对第*j*个类别的得分就是第*j*个元素： $s_j = f(x_i, W)_j$ 。针对第*i*个数据的多类SVM的损失函数定义如下：

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta)$$

**举例：**用一个例子演示公式是如何计算的。假设有3个分类，并且得到了分值  $s = [13, -7, 11]$ 。其中第一个类别是正确类别，即  $y_i = 0$ 。同时假设  $\Delta$  是10（后面会详细介绍该超参数）。上面的公式是将所有不正确分类 ( $j \neq y_i$ ) 加起来，所以我们得到两个部分：

$$L_i = \max(0, -7 - 13 + 10) + \max(0, 11 - 13 + 10)$$

可以看到第一个部分结果是0，这是因为  $[-7-13+10]$  得到的是负数，经过  $\max(0, -)$  函数处理后得到0。这一对类别分数和标签的损失值是0，这是因为正确分类的得分13与错误分类的得分-7的差为20，高于边界值10。而SVM只关心差距至少要大于10，更大的差值还是算作损失值为0。第二个部分计算  $[11-13+10]$  得到8。虽然正确分类的得分比不正确分类的得分要高 ( $13 > 11$ )，但是比10的边界值还是小了，分差只有2，这就是为什么损失值等于8。简而言之，SVM的损失函数想要正确分类类别  $y_i$  的分数比不正确类别分数高，而且至少要高  $\Delta$ 。如果不满足这点，就开始计算损失值。

换句话说，我们希望能向某些特定的权重W添加一些偏好，对其他权重则不添加，以此来消除模糊性。这一点是能够实现的，方法是向损失函数增加一个正则化惩罚（regularization penalty） $R(W)$ 部分。最常用的正则化惩罚是L2范式，L2范式通过对所有参数进行逐元素的平方惩罚来抑制大数值的权重：

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

上面的表达式中，将W中所有元素平方后求和。注意正则化函数不是数据的函数，仅基于权重。包含正则化惩罚后，就能够给出完整的多类SVM损失函数了，它由两个部分组成：数据损失（data loss），即所有样例的平均损失 $L_i$ ，以及正则化损失（regularization loss）。完整公式如下所示：

$$L = \underbrace{\frac{1}{N} \sum_i L_i}_{\text{data loss}} + \underbrace{\lambda R(W)}_{\text{regularization loss}}$$

将其展开完整公式是：

$$L = \frac{1}{N} \sum_i \sum_{j \neq y_i} [\max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + \Delta)] + \lambda \sum_k \sum_l W_{k,l}^2$$

其中，N是训练集的数据量。现在正则化惩罚添加到了损失函数里面，并用超参数 $\lambda$ 来计算其权重。该超参数无法简单确定，需要通过交叉验证来获取。

```
def L_i(x, y, W):
    """
    unvectorized version. Compute the multiclass svm loss for a single
    example (x,y)
    - x is a column vector representing an image (e.g. 3073 x 1 in CIFAR-10)
        with an appended bias dimension in the 3073-rd position (i.e. bias trick)
    - y is an integer giving index of correct class (e.g. between 0 and 9 in CIFAR-10)
    - W is the weight matrix (e.g. 10 x 3073 in CIFAR-10)
    """
    delta = 1.0 # see notes about delta later in this section
    scores = W.dot(x) # scores becomes of size 10 x 1, the scores for
    each class
    correct_class_score = scores[y]
    D = W.shape[0] # number of classes, e.g. 10
    loss_i = 0.0
    for j in xrange(D): # iterate over all wrong classes
        if j == y:
            # skip for the true class to only loop over incorrect classes
            continue
        # accumulate loss for the i-th example
        loss_i += max(0, scores[j] - correct_class_score + delta)
    return loss_i

def L_i_vectorized(x, y, W):
    """
    A faster half-vectorized implementation. half-vectorized
    """
```

```

    refers to the fact that for a single example the implementation contains
no for loops, but there is still one loop over the examples (outside this function)

"""

delta = 1.0
scores = W.dot(x)
# compute the margins for all classes in one vector operation
margins = np.maximum(0, scores - scores[y] + delta)
# on y-th position scores[y] - scores[y] canceled and gave delta.
We want
# to ignore the y-th position and only consider margin on max wrong class
margins[y] = 0
loss_i = np.sum(margins)
return loss_i

def L(X, y, W):
"""

fully-vectorized implementation :
- X holds all the training examples as columns (e.g. 3073 x 50,000 in CIFAR-10)
- y is array of integers specifying correct class (e.g. 50,000-D array)
- W are weights (e.g. 10 x 3073)
"""

# evaluate loss over all examples in X without using any for loop
s
# left as exercise to reader in the assignment

```

## 对图像均值化

```

# first: compute the image mean based on the training data
mean_image = np.mean(X_train, axis=0)
print(mean_image[:10]) # print a few of the elements
plt.figure(figsize=(4,4))
plt.imshow(mean_image.reshape((32,32,3)).astype('uint8')) # visualize the mean image
plt.show()

# second: subtract the mean image from train and test data
X_train -= mean_image

# third: append the bias dimension of ones (i.e. bias trick) so tha

```

```

t our SVM
# only has to worry about optimizing a single weight matrix W.
X_train = np.hstack([X_train, np.ones((X_train.shape[0], 1))])#水平(按列顺序)把数组给堆叠起来
X_val = np.hstack([X_val, np.ones((X_val.shape[0], 1))])

```

# 激活函数

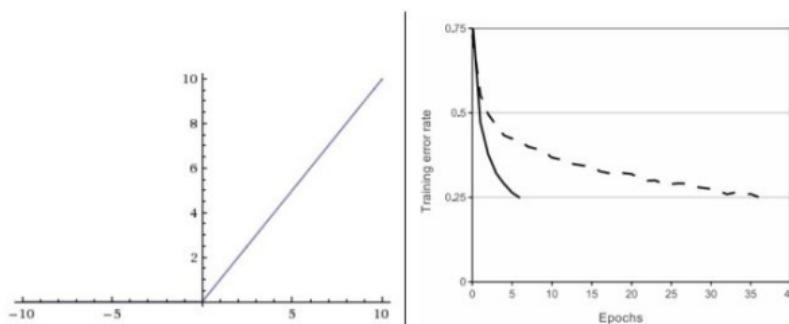
## 1. sigmoid

是因为它有两个主要缺点：

- *Sigmoid*函数饱和使梯度消失。sigmoid神经元有一个不好的特性，就是当神经元的激活在接近0或1处时会饱和：在这些区域，梯度几乎为0。回忆一下，在反向传播的时候，这个（局部）梯度将会与整个损失函数关于该单元输出的梯度相乘。因此，如果局部梯度非常小，那么相乘的结果也会接近零，这会有效地“杀死”梯度，几乎就没有信号通过神经元传到权重再到数据了。还有，为了防止饱和，必须对于权重矩阵初始化特别留意。比如，如果初始化权重过大，那么大多数神经元将会饱和，导致网络就几乎不学习了。
- *Sigmoid*函数的输出不是零中心的。这个性质并不是我们想要的，因为在神经网络后面层中的神经元得到的数据将不是零中心的。这一情况将影响梯度下降的运作，因为如果输入神经元的数据总是正数（比如在  $f = w^T x + b$  中每个元素都  $x > 0$ ），那么关于  $w$  的梯度在反向传播的过程中，将会要么全部是正数，要么全部是负数（具体依整个表达式  $f$  而定）。这将导致梯度下降权重更新时出现Z字型的下降。然而，可以看到整个批量的数据的梯度被加起来后，对于权重的最终更新将会有不同的正负，这样就从一定程度上减轻了这个问题。因此，该问题相对于上面的神经元饱和问题来说只是个小麻烦，没有那么严重。

## 2. Tanh

**Tanh。** *tanh*非线性函数图像如上图右边所示。它将实数值压缩到[-1,1]之间。和sigmoid神经元一样，它也存在饱和问题，但是和sigmoid神经元不同的是，它的输出是零中心的。因此，在实际操作中，*tanh*非线性函数比*sigmoid*非线性函数更受欢迎。注意*tanh*神经元是一个简单放大的*sigmoid*神经元，具体说来就是： $\tanh(x) = 2\sigma(2x) - 1$ 。



左边是ReLU (校正线性单元：Rectified Linear Unit) 激活函数，当  $x = 0$  时函数值为0。当  $x > 0$  函数的斜率为1。右边是从 Krizhevsky 等的论文中截取的图表，指明使用ReLU比使用 *tanh*的收敛速度快。

### 3. Relu

**ReLU。** 在近些年ReLU变得非常流行。它的函数公式是  $f(x) = \max(0, x)$ 。换句话说，这个激活函数就是一个关于0的阈值（如上图左侧）。使用ReLU有以下一些优缺点：

- 优点：相较于sigmoid和tanh函数，ReLU对于随机梯度下降的收敛有巨大的加速作用（Krizhevsky等的论文指出有6倍之多）。据称这是由它的线性，非饱和的公式导致的。
- 优点：sigmoid和tanh神经元含有指数运算等耗费计算资源的操作，而ReLU可以简单地通过对一个矩阵进行阈值计算得到。
- 缺点：在训练的时候，ReLU单元比较脆弱并且可能“死掉”。举例来说，当一个很大的梯度流过ReLU的神经元的时候，可能会导致梯度更新到一种特别的状态，在这种状态下神经元将无法被其他任何数据点再次激活。如果这种情况发生，那么从此所以流过这个神经元的梯度将都变成0。也就是说，这个ReLU单元在训练中将不可逆转的死亡，因为这导致了数据多样化的丢失。例如，如果学习率设置得太高，可能会发现网络中40%的神经元都会死掉（在整个训练集中这些神经元都不会被激活）。通过合理设置学习率，这种情况的发生概率会降低。

### 4. Leaky ReLU

**Leaky ReLU。** Leaky ReLU是为解决“ReLU死亡”问题的尝试。ReLU中当 $x < 0$ 时，函数值为0。而Leaky ReLU则是给出一个很小的负数梯度值，比如0.01。所以其函数公式为  $f(x) = 1(x < 0)(\alpha x) + 1(x \geq 0)(x)$  其中 $\alpha$ 是一个小的常量。有些研究者的论文指出这个激活函数表现很不错，但是其效果并不是很稳定。Kaiming He等人在2015年发布的论文Delving Deep into Rectifiers中介绍了一种新方法PReLU，把负区间上的斜率当做每个神经元中的一个参数。然而该激活函数在在不同任务中均有益处的一致性并没有特别清晰。

### 5. Maxout

**Maxout。** 一些其他类型的单元被提了出来，它们对于权重和数据的内积结果不再使用  $f(w^T x + b)$  函数形式。一个相关的流行选择是Maxout（最近由Goodfellow等发布）神经元。Maxout是对ReLU和leaky ReLU的一般化归纳，它的函数是： $\max(w_1^T x + b_1, w_2^T x + b_2)$ 。ReLU和Leaky ReLU都是这个公式的特殊情况（比如ReLU就是当  $w_1, b_1 = 0$  的时候）。这样Maxout神经元就拥有ReLU单元的所有优点（线性操作和不饱和），而没有它的缺点（死亡的ReLU单元）。然而和ReLU对比，它每个神经元的参数数量增加了一倍，这就导致整体参数的数量激增。

以上就是一些常用的神经元及其激活函数。最后需要注意一点：在同一个网络中混合使用不同类型的神经元是非常少见的，虽然没有什么根本性问题来禁止这样做。

一句话：“那么该用那种呢？”用ReLU非线性函数。注意设置好学习率，或许可以监控你的网络中死亡的神经元占的比例。如果单元死亡问题困扰你，就试试Leaky ReLU或者Maxout，不要再用sigmoid了。也可以试试tanh，但是其效果应该不如ReLU或者Maxout。

## 权重初始化

**使用 $1/\sqrt{n}$ 校准方差。**上面做法存在一个问题，随着输入数据量的增长，随机初始化的神经元的输出数据的分布中的方差也在增大。我们可以除以输入数据量的平方根来调整其数值范围，这样神经元输出的方差就归一化到1了。也就是说，建议将神经元的权重向量初始化为：  
 $w = np.random.randn(n) / \sqrt{n}$ 。其中n是输入数据的数量。这样就保证了网络中所有神经元起始时有近似同样的输出分布。实践经验证明，这样做可以提高收敛的速度。

1. 代码为 $w = np.random.randn(n) * \sqrt{2.0/n}$ 。这个形式是神经网络算法使用ReLU神经元时的当前最佳推荐。, n是神经元输入数
- 

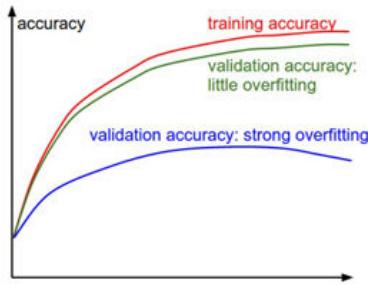
## 图片 可视化

```
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
num_classes = len(classes)
samples_per_class = 7
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y)
    idxs = np.random.choice(idxs, samples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(X_train[idx].astype('uint8'))
        plt.axis('off')
        if i == 0:
            plt.title(cls)
plt.show()
```

## jupyter notebook中重新加载

```
另一个设定 可以使notebook自动重载外部python 模块. [点击此处查看详情][4]
# 也就是说, 当从外部文件引入的函数被修改之后, 在notebook中调用这个函数, 得到的
# 被改过的函数.
%load_ext autoreload
%autoreload 2
```

# 训练集和验证集准确率



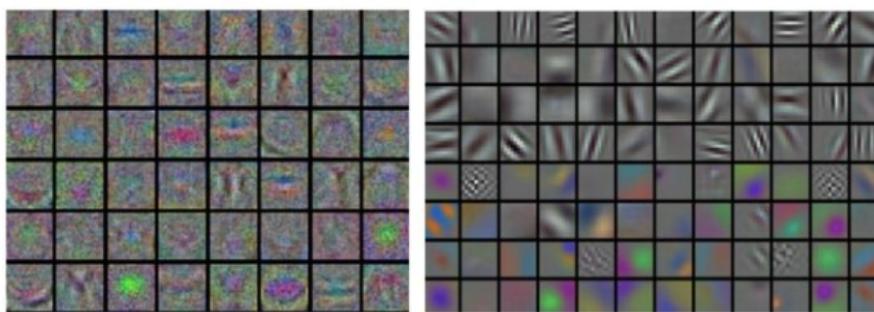
在训练集准确率和验证集准确率中间的空隙指明了模型过拟合的程度。在图中，蓝色的验证集曲线显示相较于训练集，验证集的准确率低了很多，这就说明模型有很强的过拟合。遇到这种情况，就应该增大正则化强度（更强的L2权重惩罚，更多的随机失活等）或收集更多的数据。另一种可能就是验证集曲线和训练集曲线如影随形，这种情况说明你的模型容量还不够大：应该通过增加参数数量让模型容量更大些。

## 特征可视化

1. 学习到的权重可视化，从图像可以看出，权重是用于对原图像进行特征提取的工具，与原图像关系很大。很朴素的思想，在分类器权重向量上投影最大的向量得分应该最高，训练样本得到的权重向量，最好的结果就是训练样本提取出来的共性的方向，类似于一种模板或者过滤器。

### 第一层可视化

最后，如果数据是图像像素数据，那么把第一层特征可视化会有帮助：



将神经网络第一层的权重可视化的例子。左图中的特征充满了噪音，这暗示了网络可能出现了问题：网络没有收敛，学习率设置不恰当，正则化惩罚的权重过低。右图的特征不错，平滑，干净而且种类繁多，说明训练过程进行良好。

```

1 #对于每一类，可视化学习到的权重
2 #依赖于你对学习权重和正则化强度的选择，这些可视化效果或者很明显或者不明显。
3 w = best_svm.W[:-1,:]
4 w = w.reshape(32, 32, 3, 10)
5 w_min, w_max = np.min(w), np.max(w)
6 classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship',
7 for i in range(10):
8     plt.subplot(2, 5, i + 1)
9
10    # Rescale the weights to be between 0 and 255
11    wimg = 255.0 * (w[:, :, :, i].squeeze() - w_min) / (w_max - w_min)
12    plt.imshow(wimg.astype('uint8'))
13    plt.axis('off')
14    plt.title(classes[i])

```

## 参数更新

### 1. 动量 ( Momentum ) 更新

**动量 ( Momentum )** 更新是另一个方法，这个方法在深度网络上几乎总能得到更好的收敛速度。该方法可以看成是从物理角度上对于最优化问题得到的启发。损失值可以理解为是山的高度（因此高度势能是  $U = mgh$ ，所以有  $U \propto h$ ）。用随机数字初始化参数等同于在某个位置给质点设定初始速度为0。这样最优化过程可以看做是模拟参数向量（即质点）在地形上滚动的过程。

因为作用于质点的力与梯度的潜在能量 ( $F = -\nabla U$ ) 有关，质点所受的力就是损失函数的**(负)梯度**。还有，因为  $F = ma$ ，所以在这种观点下(负)梯度与质点的加速度是成比例的。注意这个理解和上面的随机梯度下降 (SGD) 是不同的，在普通版本中，梯度直接影响位置。而在这种版本的更新中，物理观点建议梯度只是影响速度，然后速度再影响位置：

```

# 动量更新
v = mu * v - learning_rate * dx # 与速度融合
x += v # 与位置融合

```

在这里引入了一个初始化为0的变量v和一个超参数mu。说得不恰当一点，这个变量 (mu) 在最优化的过程中被看做**动量**（一般值设为0.9），但其物理意义与摩擦系数更一致。这个变量有效地抑制了速度，降低了系统的动能，不然质点在山底永远不会停下来。通过交叉验证，这个参数通常设为[0.5, 0.9, 0.95, 0.99]中的一个。和学习率随着时间退火（下文有讨论）类似，动量随时间变化的设置有时能略微改善最优化的效果，其中动量在学习过程的后阶段会上升。一个典型的设置是刚开始将动量设为0.5而在后面的多个周期 (epoch) 中慢慢提升到0.99。

### 2. Nesterov动量

**Nesterov动量**与普通动量有些许不同，最近变得比较流行。在理论上对于凸函数它能得到更好的收敛，在实践中也确实比标准动量表现更好一些。

Nesterov动量的核心思路是，当参数向量位于某个位置 $x$ 时，观察上面的动量更新公式可以发现，动量部分（忽视带梯度的第二个部分）会通过 $\mu * v$ 稍微改变参数向量。因此，如果要计算梯度，那么可以将未来的近似位置 $x + \mu * v$ 看做是“向前看”，这个点在我们一会儿要停止的位置附近。因此，计算 $x + \mu * v$ 的梯度而不是“旧”位置 $x$ 的梯度就有意义了。

也就是说，添加一些注释后，实现代码如下：

```
x_ahead = x + mu * v
# 计算dx_ahead(在x_ahead处的梯度，而不是在x处的梯度)
v = mu * v - learning_rate * dx_ahead
x += v
```

### 3. Adagrad

**Adagrad**是一个由Duchi等提出的适应性学习率算法

```
# 假设有梯度和参数向量x
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

注意，变量**cache**的尺寸和梯度矩阵的尺寸是一样的，还跟踪了每个参数的梯度的平方和。这个一会儿将用来归一化参数更新步长，归一化是逐元素进行的。注意，接收到高梯度值的权重更新的效果被减弱，而接收到低梯度值的权重的更新效果将会增强。有趣的是平方根的操作非常重要，如果去掉，算法的表现将会糟糕很多。用于平滑的式子**eps**（一般设为1e-4到1e-8之间）是防止出现除以0的情况。Adagrad的一个缺点是，在深度学习中单调的学习率被证明通常过于激进且过早停止学习。

### 4. RMSprop

**RMSprop**。是一个非常高效，但没有公开发表的适应性学习率方法。有趣的是，每个使用这个方法的人在他们的论文中都引用自Geoff Hinton的Coursera课程的第六课的第29页PPT。这个方法用一种很简单的方式修改了Adagrad方法，让它不那么激进，单调地降低了学习率。具体说来，就是它使用了一个梯度平方的滑动平均：

```
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

在上面的代码中，**decay\_rate**是一个超参数，常用的值是[0.9,0.99,0.999]。其中**x+=**和Adagrad中是一样的，但是**cache**变量是不同的。因此，RMSProp仍然是基于梯度的大小来对每个权重的学习率进行修改，这同样效果不错。但是和Adagrad不同，其更新不会让学习率单调变小。

### 5. Adam

**Adam。** Adam是最近才提出的一种更新方法，它看起来像是RMSProp的动量版。简化的代码是下面这样：

```
m = beta1*m + (1-beta1)*dx  
v = beta2*v + (1-beta2)*(dx**2)  
x += - learning_rate * m / (np.sqrt(v) + eps)
```

注意这个更新方法看起来真的和RMSProp很像，除了使用的是平滑版的梯度m，而不是用的原始梯度向量dx。论文中推荐的参数值 $\text{eps}=1e-8$ ,  $\text{beta1}=0.9$ ,  $\text{beta2}=0.999$ 。在实际操作中，我们推荐Adam作为默认的算法，一般而言跑起来比RMSProp要好一点。但是也可以试试SGD+Nesterov动量。完整的Adam更新算法也包含了一个偏置(bias)矫正机制，因为m,v两个矩阵初始为0，在没有完全热身之前存在偏差，需要采取一些补偿措施。建议读者可以阅读论文查看细节，或者课程的PPT。

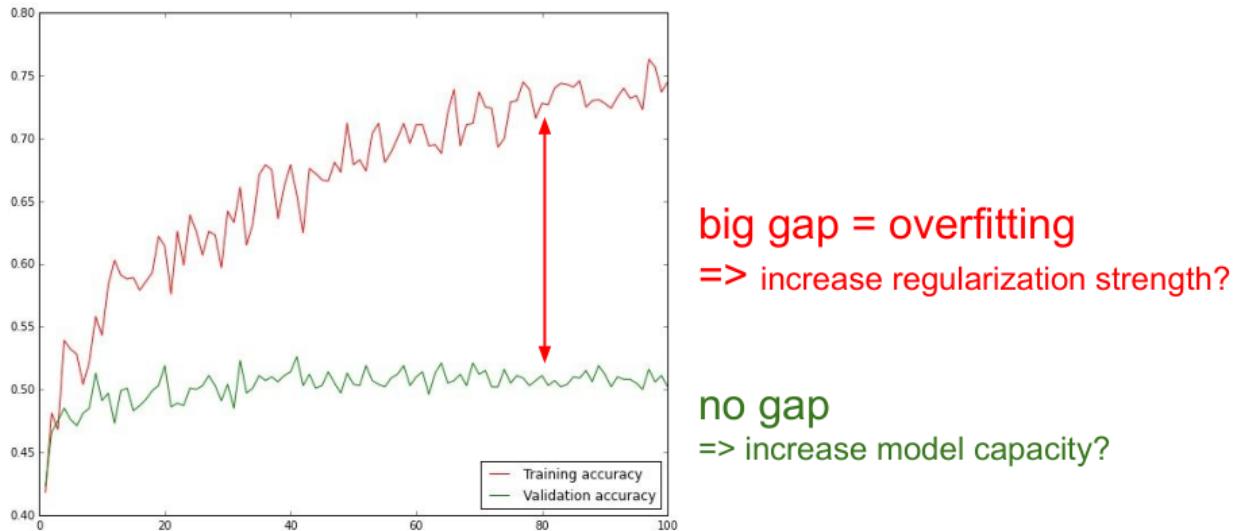
## 学习率退火

- **随步数衰减**：每进行几个周期就根据一些因素降低学习率。典型的值是每过5个周期就将学习率减少一半，或者每20个周期减少到之前的0.1。这些数值的设定是严重依赖具体问题和模型的选择的。在实践中可能看见这么一种经验做法：使用一个固定的学习率来进行训练的同时观察验证集错误率，每当验证集错误率停止下降，就乘以一个常数(比如0.5)来降低学习率。
- **指数衰减**。数学公式是  $\alpha = \alpha_0 e^{-kt}$ ，其中  $\alpha_0, k$  是超参数，t 是迭代次数(也可以使用周期作为单位)。
- **1/t衰减**的数学公式是  $\alpha = \alpha_0 / (1 + kt)$ ，其中  $\alpha_0, k$  是超参数，t 是迭代次数。

## 调参

**loss not going down:**  
learning rate too low  
**loss exploding:**  
learning rate too high

=> Rough range for learning rate we should be cross-validating is somewhere [1e-3 ... 1e-5]



## Summary

We looked in detail at:

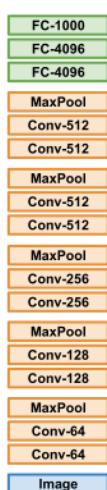
- Activation Functions (use ReLU)
- Data Preprocessing (images: subtract mean)
- Weight Initialization (use Xavier init)
- Batch Normalization (use)
- Babysitting the Learning process
- Hyperparameter Optimization  
(random sample hyperparams, in log space when appropriate)

## TLDRs

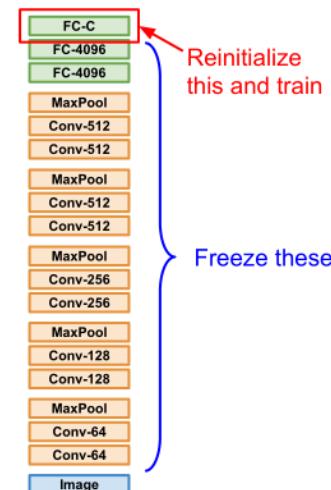
# 迁移学习

## Transfer Learning with CNNs

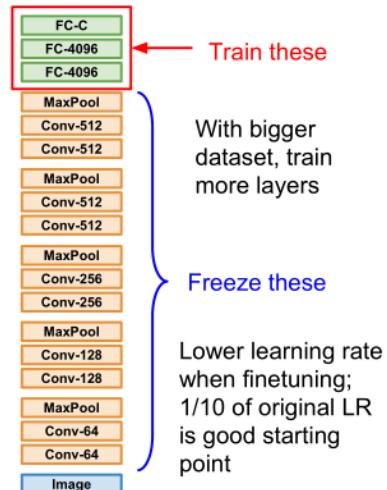
### 1. Train on Imagenet



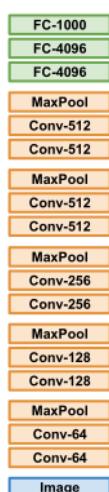
### 2. Small Dataset (C classes)



### 3. Bigger dataset



Udonian et al., "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014  
Razavian et al., "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014



More specific

More generic

	<b>very similar dataset</b>	<b>very different dataset</b>
<b>very little data</b>	Use Linear Classifier on top layer	You're in trouble... Try linear classifier from different stages
<b>quite a lot of data</b>	Finetune a few layers	Finetune a larger number of layers

## 网络架构

### 1. AlexNet

Input: 227x227x3 images

**First layer (CONV1):** 96 11x11 filters applied at stride 4

=>

Output volume **[55x55x96]**

Parameters:  $(11 \times 11 \times 3) \times 96 = 35K$

Input: 227x227x3 images

After CONV1: 55x55x96

**Second layer (POOL1):** 3x3 filters applied at stride 2

Q: what is the output volume size? Hint:  $(55-3)/2+1 = 27$

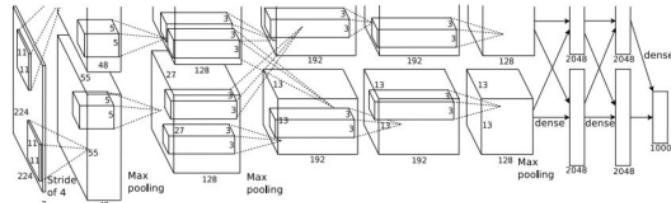
Input: 227x227x3 images  
After CONV1: 55x55x96

**Second layer (POOL1):** 3x3 filters applied at stride 2  
Output volume: 27x27x96  
Parameters: 0!

## Case Study: AlexNet

[Krizhevsky et al. 2012]

Full (simplified) AlexNet architecture:  
[227x227x3] INPUT  
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0  
[27x27x96] MAX POOL1: 3x3 filters at stride 2  
[27x27x96] NORM1: Normalization layer  
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2  
[13x13x256] MAX POOL2: 3x3 filters at stride 2  
[13x13x256] NORM2: Normalization layer  
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1  
[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1  
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1  
[6x6x256] MAX POOL3: 3x3 filters at stride 2  
[4096] FC6: 4096 neurons  
[4096] FC7: 4096 neurons  
[1000] FC8: 1000 neurons (class scores)



### Details/Retrospectives:

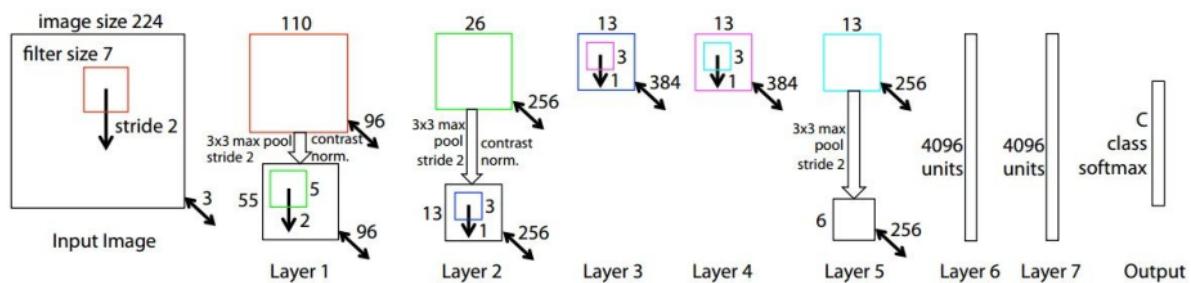
- first use of ReLU
- used Norm layers (not common anymore)
- heavy data augmentation
- dropout 0.5
- batch size 128
- SGD Momentum 0.9
- Learning rate 1e-2, reduced by 10 manually when val accuracy plateaus
- L2 weight decay 5e-4
- 7 CNN ensemble: 18.2% -> 15.4%

Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

## 2. ZFNet

### ZFNet

[Zeiler and Fergus, 2013]



TODO: remake figure

AlexNet but:

CONV1: change from (11x11 stride 4) to (7x7 stride 2)

CONV3,4,5: instead of 384, 384, 256 filters use 512, 1024, 512

ImageNet top 5 error: 16.4% -> 11.7%

## 3. VGG16

INPUT: [224x224x3] memory:  $224 \times 224 \times 3 = 150K$  params: 0 (not counting biases)  
 CONV3-64: [224x224x64] memory:  $224 \times 224 \times 64 = 3.2M$  params:  $(3 \times 3 \times 3) \times 64 = 1,728$   
 CONV3-64: [224x224x64] memory:  $224 \times 224 \times 64 = 3.2M$  params:  $(3 \times 3 \times 64) \times 64 = 36,864$   
 POOL2: [112x112x64] memory:  $112 \times 112 \times 64 = 800K$  params: 0  
 CONV3-128: [112x112x128] memory:  $112 \times 112 \times 128 = 1.6M$  params:  $(3 \times 3 \times 64) \times 128 = 73,728$   
 CONV3-128: [112x112x128] memory:  $112 \times 112 \times 128 = 1.6M$  params:  $(3 \times 3 \times 128) \times 128 = 147,456$   
 POOL2: [56x56x128] memory:  $56 \times 56 \times 128 = 400K$  params: 0  
 CONV3-256: [56x56x256] memory:  $56 \times 56 \times 256 = 800K$  params:  $(3 \times 3 \times 128) \times 256 = 294,912$   
 CONV3-256: [56x56x256] memory:  $56 \times 56 \times 256 = 800K$  params:  $(3 \times 3 \times 256) \times 256 = 589,824$   
 CONV3-256: [56x56x256] memory:  $56 \times 56 \times 256 = 800K$  params:  $(3 \times 3 \times 256) \times 256 = 589,824$   
 POOL2: [28x28x256] memory:  $28 \times 28 \times 256 = 200K$  params: 0  
 CONV3-512: [28x28x512] memory:  $28 \times 28 \times 512 = 400K$  params:  $(3 \times 3 \times 256) \times 512 = 1,179,648$   
 CONV3-512: [28x28x512] memory:  $28 \times 28 \times 512 = 400K$  params:  $(3 \times 3 \times 512) \times 512 = 2,359,296$   
 CONV3-512: [28x28x512] memory:  $28 \times 28 \times 512 = 400K$  params:  $(3 \times 3 \times 512) \times 512 = 2,359,296$   
 POOL2: [14x14x512] memory:  $14 \times 14 \times 512 = 100K$  params: 0  
 CONV3-512: [14x14x512] memory:  $14 \times 14 \times 512 = 100K$  params:  $(3 \times 3 \times 512) \times 512 = 2,359,296$   
 CONV3-512: [14x14x512] memory:  $14 \times 14 \times 512 = 100K$  params:  $(3 \times 3 \times 512) \times 512 = 2,359,296$   
 CONV3-512: [14x14x512] memory:  $14 \times 14 \times 512 = 100K$  params:  $(3 \times 3 \times 512) \times 512 = 2,359,296$   
 POOL2: [7x7x512] memory:  $7 \times 7 \times 512 = 25K$  params: 0  
 FC: [1x1x4096] memory: 4096 params:  $77 \times 512 \times 4096 = 102,760,448$   
 FC: [1x1x4096] memory: 4096 params:  $4096 \times 4096 = 16,777,216$   
 FC: [1x1x1000] memory: 1000 params:  $4096 \times 1000 = 4,096,000$

Note:

Most memory is in early CONV

Most params are in late FC

TOTAL memory:  $24M * 4$  bytes  $\approx 96MB / \text{image}$  (only forward!  $\sim 2$  for bwd)  
 TOTAL params: 138M parameters

## ImageNet Large Scale Visual Recognition Challenge (ILSVRC) winners

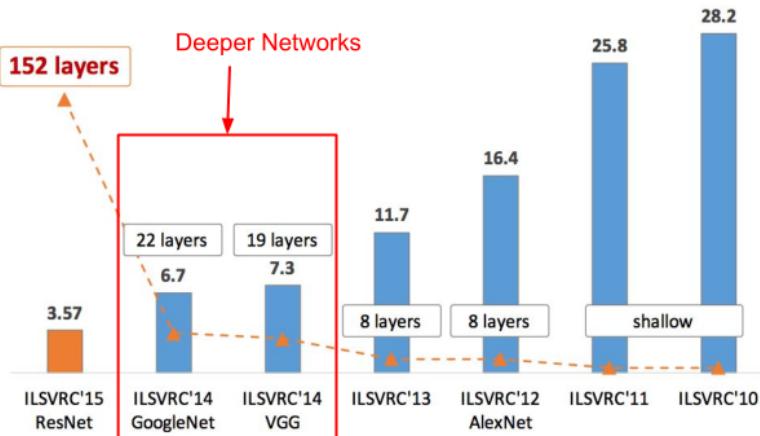


Figure copyright Kaiming He, 2016. Reproduced with permission.

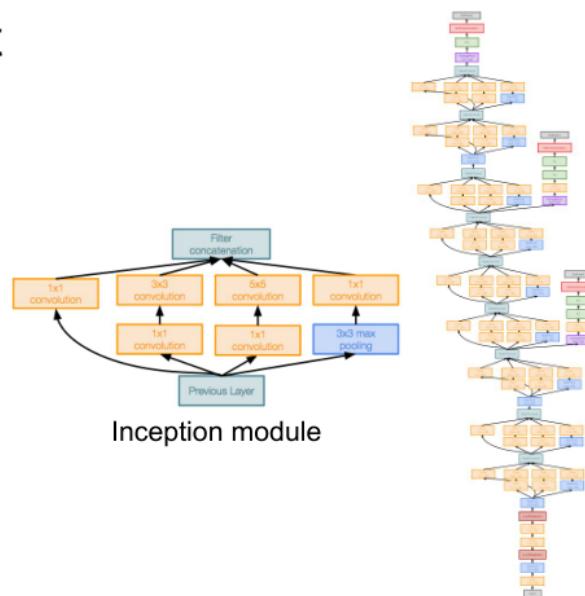
## 4. GoogLeNet

### Case Study: GoogLeNet

[Szegedy et al., 2014]

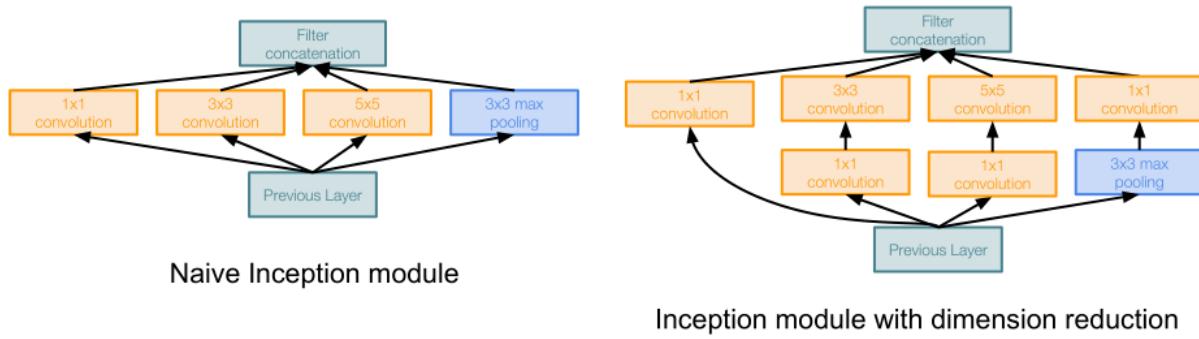
Deeper networks, with computational efficiency

- 22 layers
- Efficient “Inception” module
- No FC layers
- Only 5 million parameters!  
12x less than AlexNet
- ILSVRC'14 classification winner  
(6.7% top 5 error)



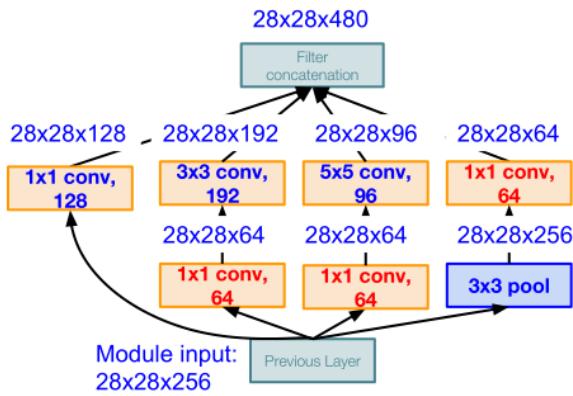
# Case Study: GoogLeNet

[Szegedy et al., 2014]



# Case Study: GoogLeNet

[Szegedy et al., 2014]



Inception module with dimension reduction

Using same parallel layers as naive example, and adding “1x1 conv, 64 filter” bottlenecks:

#### Conv Ops:

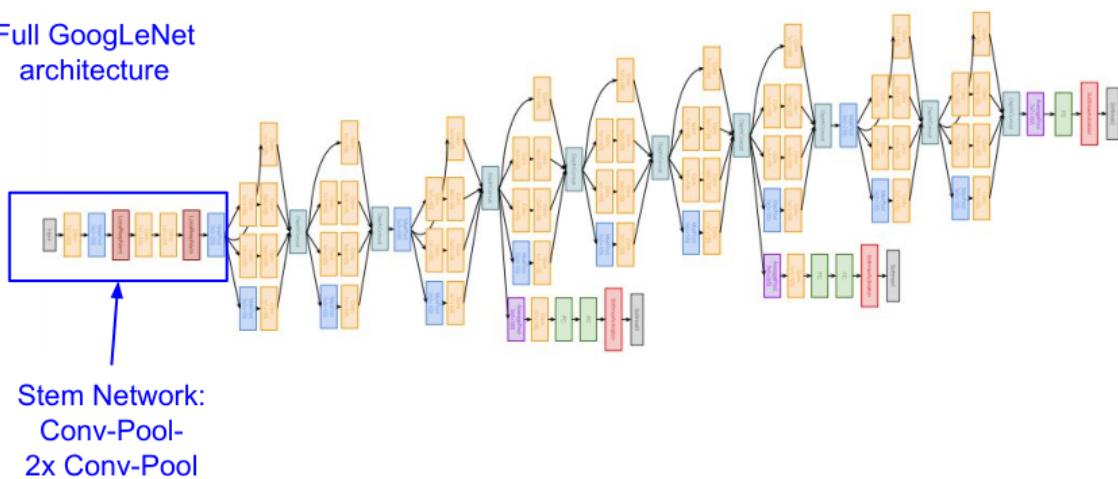
[1x1 conv, 64] 28x28x64x1x1x256  
[1x1 conv, 64] 28x28x64x1x1x256  
[1x1 conv, 128] 28x28x128x1x1x256  
[3x3 conv, 192] 28x28x192x3x3x64  
[5x5 conv, 96] 28x28x96x5x5x64  
[1x1 conv, 64] 28x28x64x1x1x256  
**Total: 358M ops**

Compared to 854M ops for naive version  
Bottleneck can also reduce depth after pooling layer

# Case Study: GoogLeNet

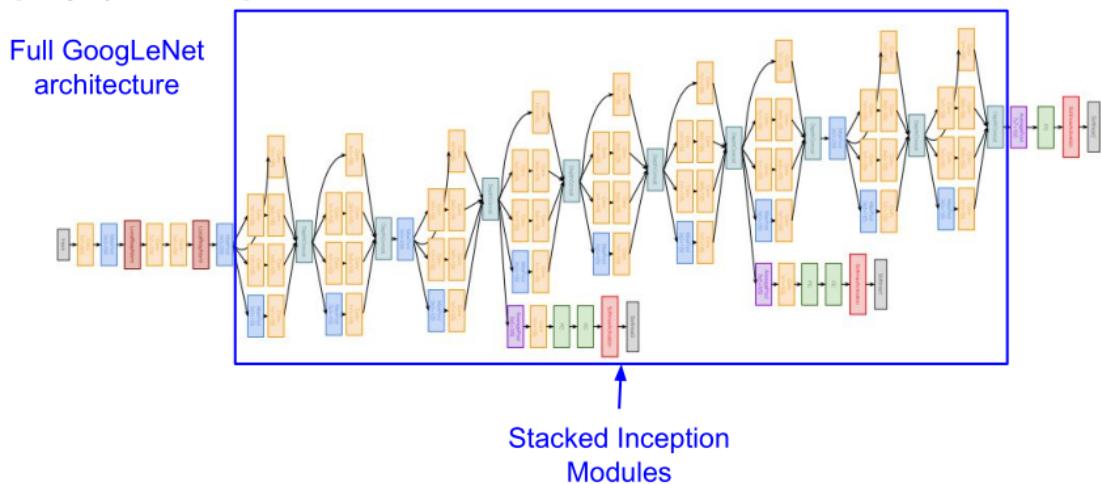
[Szegedy et al., 2014]

Full GoogLeNet architecture



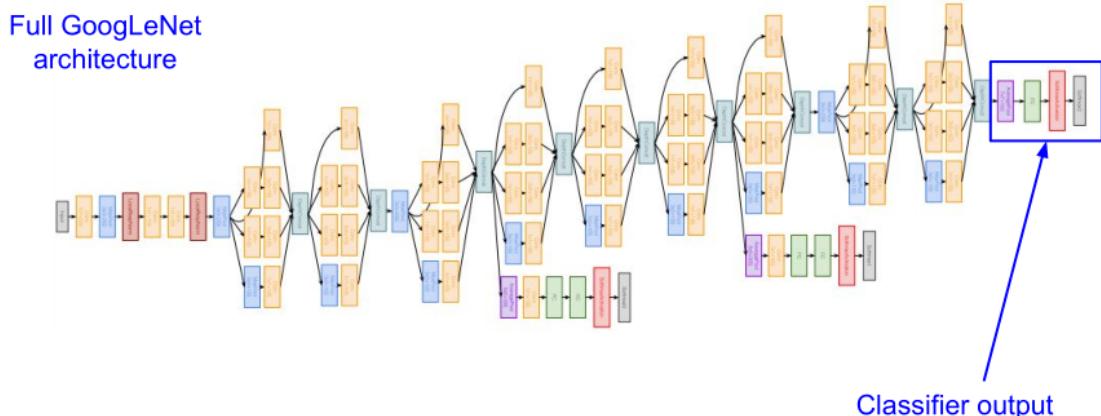
# Case Study: GoogLeNet

[Szegedy et al., 2014]



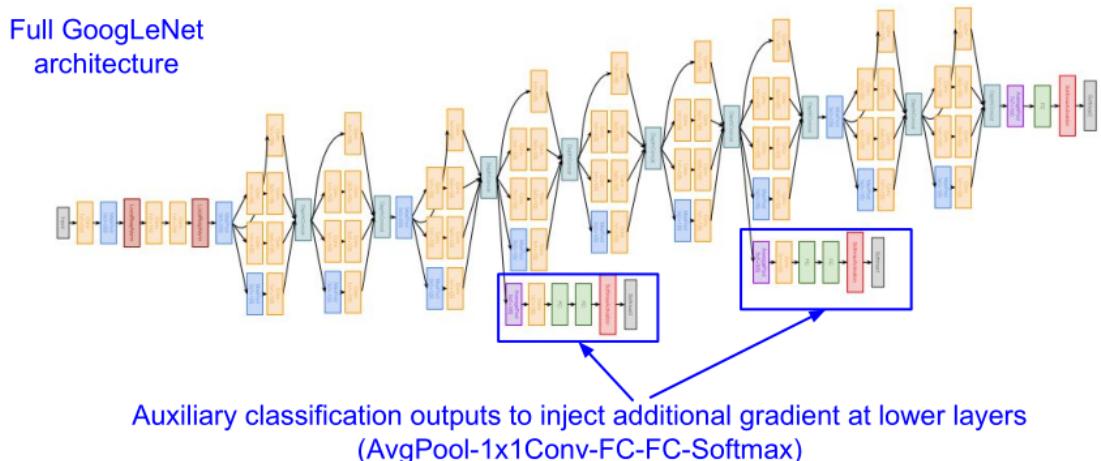
# Case Study: GoogLeNet

[Szegedy et al., 2014]



# Case Study: GoogLeNet

[Szegedy et al., 2014]

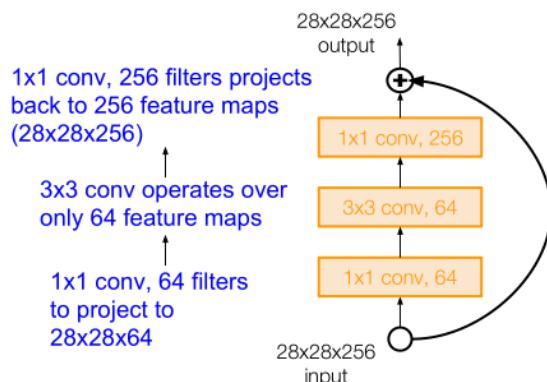


## 5. ResNet

## Case Study: ResNet

[He et al., 2015]

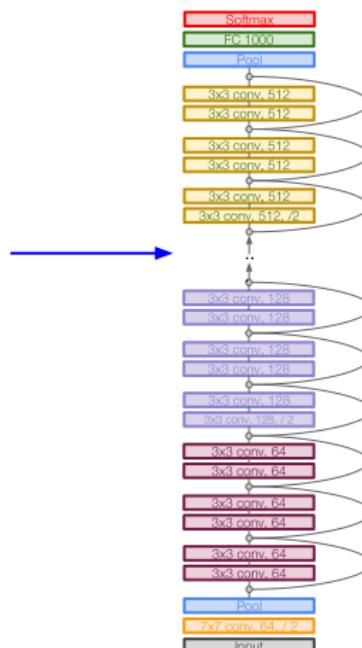
For deeper networks  
(ResNet-50+), use “bottleneck”  
layer to improve efficiency  
(similar to GoogLeNet)



## Case Study: ResNet

[He et al., 2015]

Total depths of 34, 50, 101, or 152 layers for ImageNet



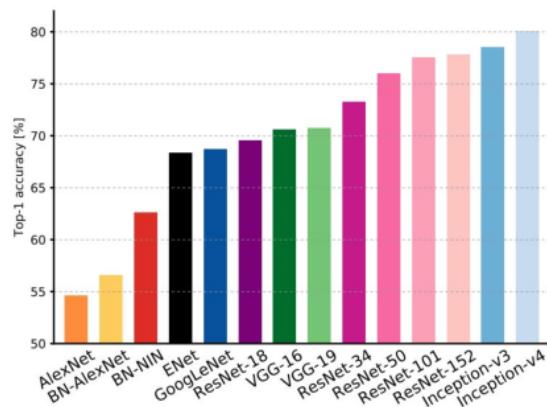
## Case Study: ResNet

[He et al., 2015]

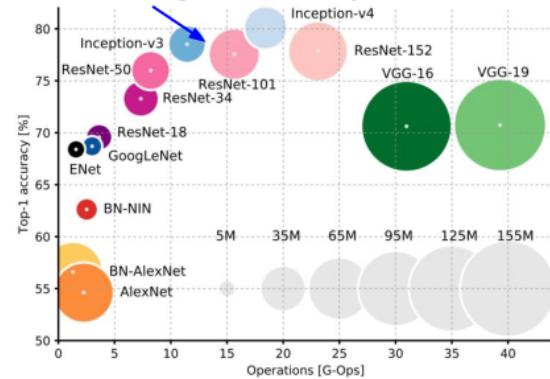
## Training ResNet in practice:

- Batch Normalization after every CONV layer
  - Xavier/2 initialization from He et al.
  - SGD + Momentum (0.9)
  - Learning rate: 0.1, divided by 10 when validation error plateaus
  - Mini-batch size 256
  - Weight decay of 1e-5
  - No dropout used

## Comparing complexity...



ResNet:  
Moderate efficiency depending on model, highest accuracy



An Analysis of Deep Neural Network Models for Practical Applications, 2017.

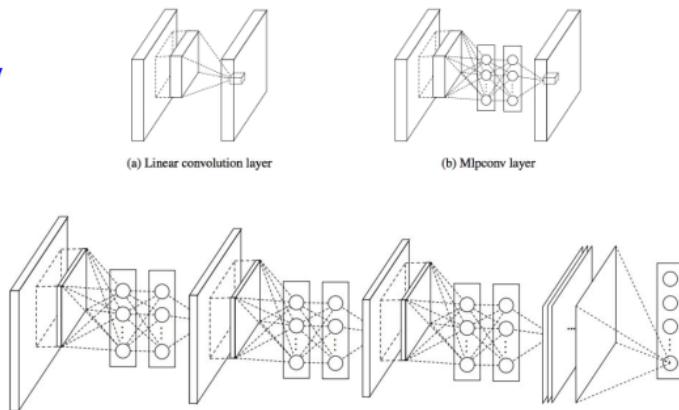
- 圆形代表所使用的存储空间，越大代表内存越大

## 6. Other architectures

### Network in Network (NiN)

[Lin et al. 2014]

- Mlpconv layer with “micronetwork” within each conv layer to compute more abstract features for local patches
- Micronetwork uses multilayer perceptron (FC, i.e. 1x1 conv layers)
- Precursor to GoogLeNet and ResNet “bottleneck” layers
- Philosophical inspiration for GoogLeNet



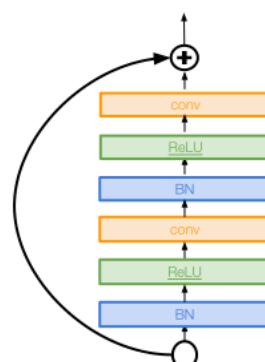
Figures copyright Lin et al., 2014. Reproduced with permission.

## Improving ResNets...

### Identity Mappings in Deep Residual Networks

[He et al. 2016]

- Improved ResNet block design from creators of ResNet
- Creates a more direct path for propagating information throughout network (moves activation to residual mapping pathway)
- Gives better performance

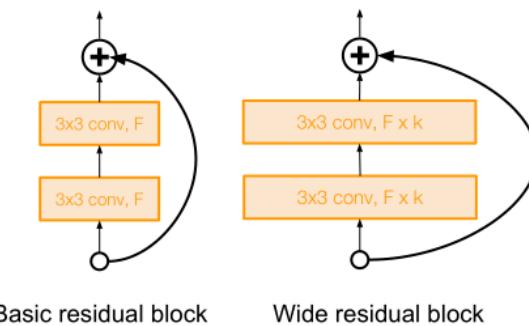


Improving ResNets...

## Wide Residual Networks

[Zagoruyko et al. 2016]

- Argues that residuals are the important factor, not depth
- Use wider residual blocks ( $F \times k$  filters instead of  $F$  filters in each layer)
- 50-layer wide ResNet outperforms 152-layer original ResNet
- Increasing width instead of depth more computationally efficient (parallelizable)

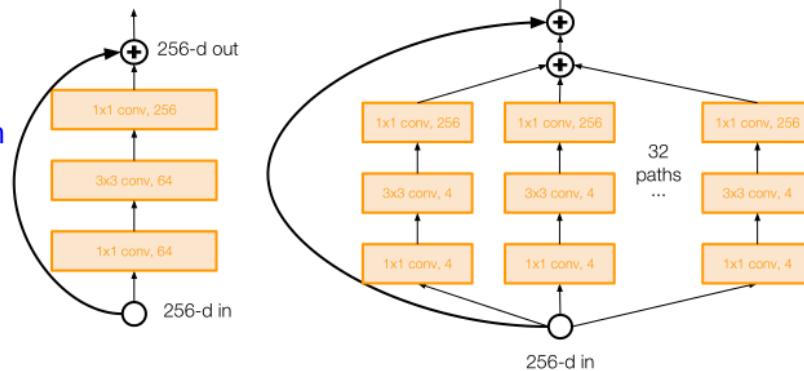


Improving ResNets...

## Aggregated Residual Transformations for Deep Neural Networks (ResNeXt)

[Xie et al. 2016]

- Also from creators of ResNet
- Increases width of residual block through multiple parallel pathways ("cardinality")
- Parallel pathways similar in spirit to Inception module

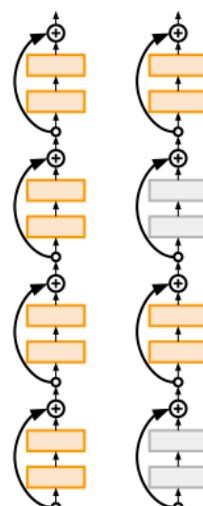


Improving ResNets...

## Deep Networks with Stochastic Depth

[Huang et al. 2016]

- Motivation: reduce vanishing gradients and training time through short networks during training
- Randomly drop a subset of layers during each training pass
- Bypass with identity function
- Use full deep network at test time

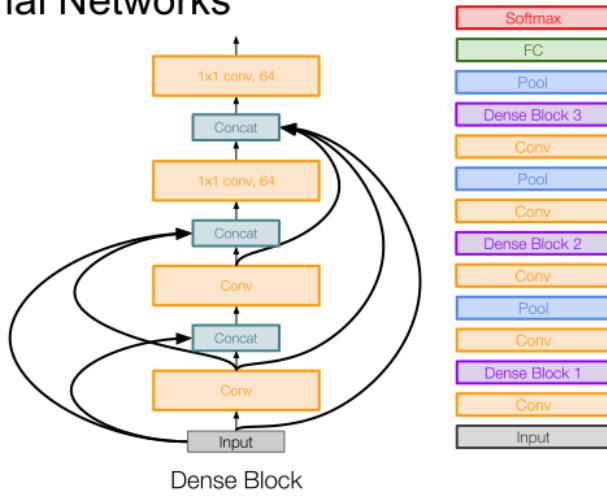


## Beyond ResNets...

### Densely Connected Convolutional Networks

[Huang et al. 2017]

- Dense blocks where each layer is connected to every other layer in feedforward fashion
- Alleviates vanishing gradient, strengthens feature propagation, encourages feature reuse



## CNN中感受视野的计算

1. 就是输出featuremap某个节点的响应对应的输入图像的区域就是感受野。

### 感受野的计算 (Receptive Field Arithmetic)

除了每个维度上特征图的个数，还需要计算每一层的感受野大小，因此我们需要了解每一层的额外信息，包括：当前感受野的尺寸 $r$ ，相邻特征之间的距离（或者jump） $j$ ，左上角（起始）特征的中心坐标 $start$ ，其中特征的中心坐标定义为其感受野的中心坐标（如上述固定大小CNN特征图所述）。假设卷积核大小 $k$ ，填充大小 $p$ ，步长大小 $s$ ，则其输出层的相关属性计算如下：

$$\begin{aligned} n_{out} &= \left\lceil \frac{n_{in} + 2p - k}{s} \right\rceil + 1 \\ j_{out} &= j_{in} * s \\ r_{out} &= r_{in} + (k - 1) * j_{in} \\ start_{out} &= start_{in} + \left( \frac{k - 1}{2} - p \right) * j_{in} \end{aligned}$$

- 公式一基于输入特征个数和卷积相关属性计算输出特征的个数
- 公式二计算输出特征图的jump，等于输入图的jump与输入特征个数（执行卷积操作时jump的个数，stride的大小）的乘积
- 公式三计算输出特征图的receptive field size，等于 $k$ 个输入特征覆盖区域 $(k - 1) * j_{in}$ 加上边界上输入特征的感受野覆盖的附加区域 $r_{in}$ 。
- 公式四计算第一个输出特征的感受野的中心位置，等于第一个输入特征的中心位置，加上第一个输入特征位置到第一个卷积核中心位置的距离 $(k - 1)/2 * j_{in}$ ，再减去填充区域大小 $p * j_{in}$ 。注意：这里都需要乘上输入特征图的jump，从而获取实际距离或间隔。

- 这里的 $j_{in}$ 是前面所有的stride乘积

## Dilated Residual Networks

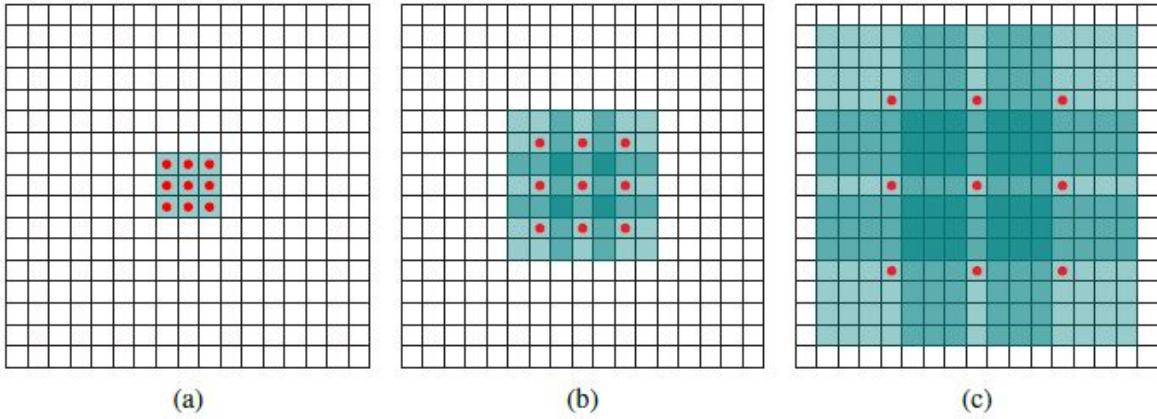


Figure 1: Systematic dilation supports exponential expansion of the receptive field without loss of resolution or coverage. (a)  $F_1$  is produced from  $F_0$  by a 1-dilated convolution; each element in  $F_1$  has a receptive field of  $3 \times 3$ . (b)  $F_2$  is produced from  $F_1$  by a 2-dilated convolution; each element in  $F_2$  has a receptive field of  $7 \times 7$ . (c)  $F_3$  is produced from  $F_2$  by a 4-dilated convolution; each element in  $F_3$  has a receptive field of  $15 \times 15$ . The number of parameters associated with each layer is identical. The receptive field grows exponentially while the number of parameters grows linearly.

上面实现过程总结来说就是，假设网络输入为 $28 \times 28$ ，我们使用padding和stride=1的卷积，卷积filter尺寸都是 $3 \times 3$ 。整个感受野的变化如上图绿色区域所示。

(a) 输入 $28 \times 28$ 基础上 $3 \times 3$ 卷积，也就是经过1-dilated处理，感受野为 $3 \times 3$ ，该操作和其他正常卷积操作一样，没有区别；

(b) 在(a)输出的基础上进行 $3 \times 3$ 卷积，经过2-dilated处理，也就是隔一个像素进行与filter点乘最后相加作为中心像素的特征值，所以感受野变为 $7 \times 7$ ；

(c) 在(b)的输出基础上进行 $3 \times 3$ 卷积，经过4-dilated处理，也就是隔三个像素进行与filter点乘最后相加作为中心像素的特征值，所以感受野变为 $15 \times 15$ ；

FCN使用池化（pooling）下采样（downsampling）来增大感受野，但随后又通过反卷积（Deconvolution）或者上采样（upsampling）来增大特征map尺寸，这样先减后增的操作会让图片特征损失很多信息。膨胀卷积可以不降低特征图的尺寸（或分辨率）而增大卷积感受野。

#### 4. Dilated Residual Networks

膨胀卷积可以应用到很多CNN上，以保护网络的空间分辨率。

——

(a)图对应 $3 \times 3$ 的1-dilated conv，和普通的卷积操作一样，(b)图对应 $3 \times 3$ 的2-dilated conv，实际的卷积kernel size还是 $3 \times 3$ ，但是空洞为1，也就是对于一个 $7 \times 7$ 的图像patch，只有9个红色的点和 $3 \times 3$ 的kernel发生卷积操作，其余的点略过。也可以理解为kernel的size为 $7 \times 7$ ，但是只有图中的9个点的权重不为0，其余都为0。可以看到虽然kernel size只有 $3 \times 3$ ，但是这个卷积的感受野已经增大到了 $7 \times 7$ （如果考虑到这个2-dilated conv的前一层是一个1-dilated conv的话，那么每个红点就是1-dilated的卷积输出，所以感受野为 $3 \times 3$ ，所以1-dilated和2-dilated合起来就能达到 $7 \times 7$ 的conv），(c)图是4-dilated conv操作，同理跟在两个1-dilated和2-dilated conv的后面，能达到 $15 \times 15$ 的感受野。对比传统的conv操作，3层 $3 \times 3$ 的卷积加起来，stride为1的话，只能达到 $(kernel-1) * layer + 1 = 7$ 的感受野，也就是和层数layer成线性关系，而dilated conv的感受野是指数级的增长。

dilated的好处是不做pooling损失信息的情况下，加大了感受野，让每个卷积输出都包含较大范围的信息。在图像需要全局信息或者语音文本需要较长的sequence信息依赖的问题中，都能很好的应用dilated conv，比如图像分割[3]、语音合成WaveNet[2]、机器翻译ByteNet[1]中。简单贴下ByteNet和WaveNet用到的dilated conv结构，可以更形象的了解dilated conv本身。

#### ByteNet

##### 1. deconv和dilated conv区别

可以形象的做个解释：

对于标准的 $k \times k$ 卷积操作，stride为s，分三种情况：

- (1)  $s > 1$ ，即卷积的同时做了downsampling，卷积后图像尺寸减小；
- (2)  $s = 1$ ，普通的步长为1的卷积，比如在tensorflow中设置padding=SAME的话，卷积的图像输入和输出有相同的尺寸大小；
- (3)  $0 < s < 1$ ，fractionally strided convolution，相当于对图像做upsampling。比如 $s=0.5$ 时，意味着在图像每个像素之间padding一个空白的像素后，stride改为1做卷积，得到的feature map尺寸增大一倍。

而dilated conv不是在像素之间padding空白的像素，而是在已有的像素上，skip掉一些像素，或者输入不变，对conv的kernel参数中插一些0的weight，达到一次卷积看到的空间范围变大的目的。

当然将普通的卷积stride步长设为大于1，也会达到增加感受野的效果，但是stride大于1就会导致downsampling，图像尺寸变小。大家可以从以上理解到deconv，dilated conv，pooling/downsampling，upsampling之间的联系与区别，欢迎留言沟通交流。

## 反卷积(Deconvolution)上采样(Upsampling)上池化(Unpooling)的区别

上采样是指将图像上采样到更高分辨率的任何技术。

最简单的方法是使用重新采样和插值。即取原始图像输入，将其重新缩放到所需的大小，然后使用插值方法（如双线性插值）计算每个点处的像素值。

在CNN上下文中，上池化通常指代最大池化的逆过程。在CNN中，最大池化操作是不可逆的，但是我们可以通过使用一组转换变量记录每个池化区域内最大值的位置来获得一个近似的逆操作结果。在反卷积（网络）中，上池化操作使用这些转换变量从前一层输入中安放这些复原物到（当前层）合适的位置，从而一定程度上保护了原有结构。

在CNN上下文中，反卷积通常用于指代卷积的逆过程，而非数学意义上真正的反卷积，这一点很重要，也很令人困惑。相比上池化，使用反卷积进行图像的上采样是可以被学习的。反卷积常被用于对CNN的输出进行上采样至原始图像分辨率。我在这里写下关于这个问题的另一个答案：

反卷积常被认为是空洞卷积或者转置卷积，这也更为恰当一些。

1. 由于上采样是指将图像上采样到更高分辨率的任何技术，因此我们可以讲：通过反卷积进行上采样。
- 

## 学习率设置

1. SGD+Momentum
  - rho=0.9 or 0.99
2. Adam
  - Adam with beta1 = 0.9, beta2 = 0.999, epsilon = 1e-8 and learning\_rate = 1e-3 or 5e-4 is a great starting point for many models
3. 正则化系数设置
  - 给 $\lambda$ 一个值（比方1.0），然后依据validation accuracy。将 $\lambda$ 增大或者减小10倍（增减10倍是粗调节，当你确定了 $\lambda$ 的合适的数量级后，比方 $\lambda = 0.01$ ,再进一步地细调节，比方调节为0.02，0.03，0.009之类。）初步尝试可以使用 1e-4 或者 1e-3