Tensorflow_CNN

<Excerpt in index | 首页摘要>
使用Tensorflow搭建CNN,熟悉一些经典的网络。
1、GoogleNet 2、ResNet 3、DenseNet 4、VGG
Github：Tensorflow
Resource:深度学习理论与实战（基于TensorFlow实现）

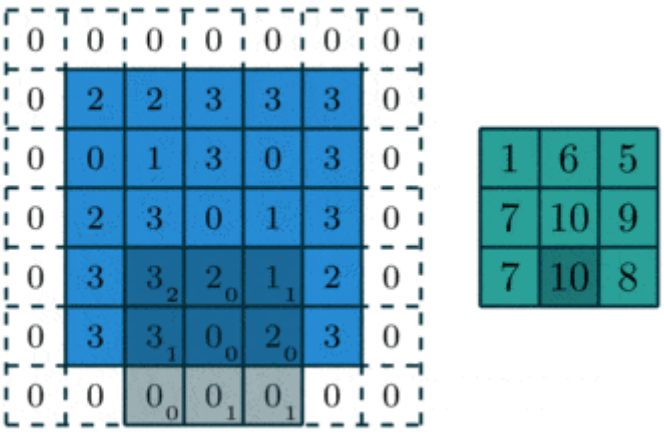
<The rest of contents | 余下全文>

卷积神经网络

1、卷积的一些基本知识

在一般的情况中, 固定ksize和stride时我们可以通过下面的公式计算出**输出结果的形状**:

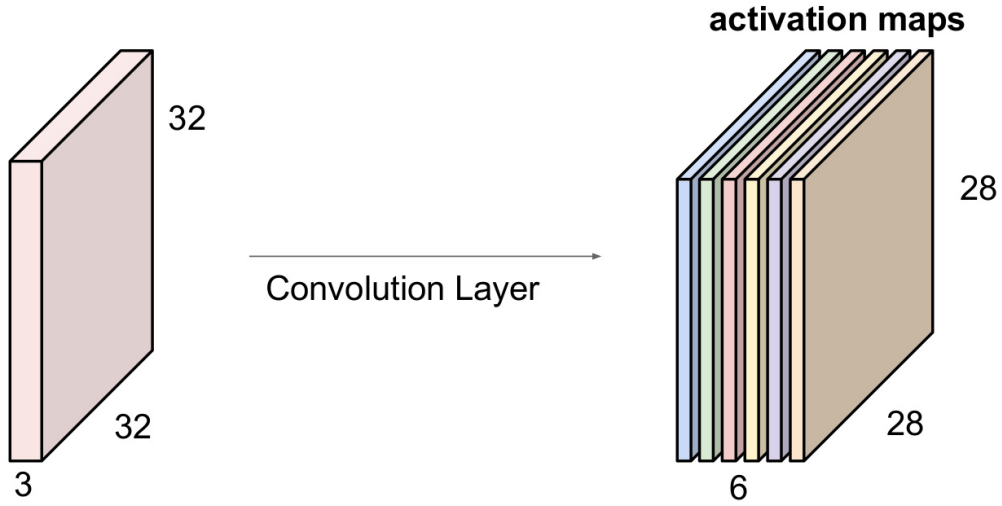
$$W_{out} = (W - K + 2P)/S + 1$$



2、卷积层

一个卷积核就可以得到一个输出, 那么多个卷积核就可以得到多个输出. 因此, 一个卷积层一般由若干个卷积核排列而成.

For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



We stack these up to get a "new image" of size 28x28x6!

上面这个图就是一个卷积层的过程, 输入是$32 \times 32 \times 3$的`RGB`通道图片, 通过6个$5 \times 5 \times 3$的卷积核得到$28 \times 28 \times 6$的输出. 现在来解读一下里面出现的数值

- 输入图片大小是3-通道的, 以后我们成为深度为`3`, 为了能够让卷积核和图片进行滑动点乘, 所以卷积核的深度也是`3`, 而卷积核的大小随意, 这里是5×5
- 通过上面计算卷积输出形状的公式, 一个$32 \times 32 \times 3$的图片经过一个$5 \times 5 \times 3$的卷积核得到一个28×28的输出
- 一共有6个卷积核, 所以最后的输出是$28 \times 28 \times 6$的形状

代码示例

卷积在原生的 `tensorflow` 中的 API 是 **`tf.nn.conv2d()`**

- 第一个参数`input`是输入, 要求一定一个形状为`(a, b, c, d)`的`tensor`,也就是通常说的4维张量. 它具有两种形式:`NHWC`, `NCHW`, 表示输入的通道在第二维或者是第四维, 在后面的参数`data_format`中可以进行选择
- 第二个参数`filter`就是参与卷积的卷积核, 要求是一个4维张量, 形状是`[height, width, in_depth, out_depth]`. 其中`height, width`表示卷积核本身的大小, `in_depth`必须和`input`的通道数保持一致, `out_depth`表示卷积核的个数
- 第三个参数`strides`是卷积核滑动的步长, 要求是一个4维张量, 第二维和第三维表示卷积核的大小, 对于`NHWC`的输入来说形状是`(1, stride_h, stride_w, 1)`, 对于`NCHW`的输入来说是`(1, 1, stride_h, stride_w)`
- 第四个参数`padding`是补洞策略, 可以选择`"SAME"`或者是`"VALID"`, 区别之后会解释
- 后面几个参数暂时不用管它, 大家可以自己探索

```
# 将图片矩阵转化为 tensor，并适配卷积输入的要求
im = tf.constant(im.reshape((1, im.shape[0], im.shape[1], 1)), name='inpu
t')

# 定义一个边缘检测算子`sobel_kernel`，并规范形状
sobel_kernel = np.array([[-1, -1, -1], [-1, 8, -1], [-1, -1, -1]], dtype=
np.float32)
sobel_kernel = tf.constant(sobel_kernel, shape=(3, 3, 1, 1))
# 进行卷积
```

```
## padding='SAME' 的卷积
edge1 = tf.nn.conv2d(im, sobel_kernel, [1, 1, 1, 1], 'SAME', name='same_c
onv')

## padding='VALID' 的卷积
edge2 = tf.nn.conv2d(im, sobel_kernel, [1, 1, 1, 1], 'VALID', name='valid
_conv')
```

3、池化层

池化层是一个下采样的操作, 用以快速地减小输入的大小同时不至于丢失重要的信息. 现在一般都有**两种池化层:**

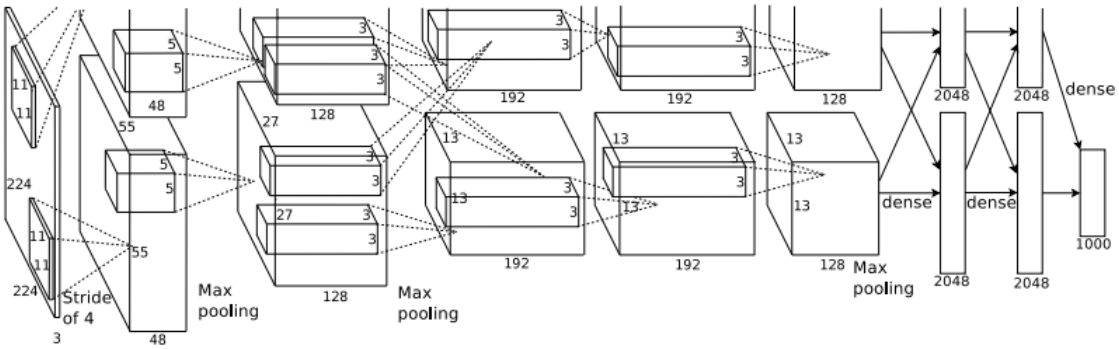
- max pooling(最大值池化层)
- average pooling(平均值池化层)

AlexNet

1、基本概念

特点:

- 更深的网络
- 数据增广
- ReLU
- dropout
- LRN



AlexNet用到训练技巧:

- 数据增广技巧来增加模型泛化能力。
- 用**ReLU代替Sigmoid**来加快SGD的收敛速度
- **Dropout**:Dropout原理类似于浅层学习算法的中集成算法，该方法通过让全连接层的神经元（该模型在前两个全连接层引入Dropout）以一定的概率失去活性（比如0.5）失活的神经元

不再参与前向和反向传播，相当于约有一半的神经元不再起作用。在测试的时候，让所有神经元的输出乘0.5。Dropout的引用，有效缓解了模型的过拟合。

2、代码示例

编写一个上层的卷积接口来调用

```
    """
    构造一个卷积层
    Args:
        x: 输入
        ksize: 卷积核的大小，一个长度为2的`list`，例如[3, 3]
        output_depth: 卷积核的个数
        strides: 卷积核移动的步长，一个长度为2的`list`，例如[2, 2]
        padding: 卷积核的补0策略
        act: 完成卷积后的激活函数，默认是`tf.nn.relu`
        scope: 这一层的名称(可选)
        reuse: 是否复用

    Return:
        out: 卷积层的结果
    """
    # 这里默认数据是NHWC输入的
    in_depth = x.get_shape().as_list()[-1]
    with tf.variable_scope(scope, reuse=reuse):
        # 先构造卷积核
        shape = ksize + [in_depth, out_depth]
        with tf.variable_scope('kernel'):
            kernel = variable_weight(shape)
        strides = [1, strides[0], strides[1], 1]
        # 生成卷积
        conv = tf.nn.conv2d(x, kernel, strides, padding, name='conv')
        # 构造偏置
        with tf.variable_scope('bias'):
            bias = variable_bias([out_depth])
        # 和偏置相加
        preact = tf.nn.bias_add(conv, bias)
        # 添加激活层
        out = act(preact)
        return out
```

搭建AlexNet

```
def alexnet(inputs, reuse=None):
    """构建 Alexnet 的前向传播
    Args:
```

```
        inpus: 输入
        reuse: 是否需要重用
    Return:
        net: alexnet的结果
    """
    # 首先我们声明一个变量域`AlexNet`
    with tf.variable_scope('AlexNet', reuse=reuse):
        # 第一层是 5x5 的卷积，卷积核的个数是64，步长是 1x1，padding是`VALID`
        net = conv(inputs, [5, 5], 64, [1, 1], padding='VALID', scope='co
nv1')
        # 第二层是 3x3 的池化，步长是 2x2，padding是`VALID`
        net = max_pool(net, [3, 3], [2, 2], padding='VALID',
name='pool1')
        # 第三层是 5x5 的卷积，卷积核的个数是64，步长是 1x1，padding是`VALID`
        net = conv(net, [5, 5], 64, [1, 1], scope='conv2')
        # 第四层是 3x3 的池化，步长是 2x2，padding是`VALID`
        net = max_pool(net, [3, 3], [2, 2], padding='VALID',
name='pool2')
        # 将矩阵拉长成向量
        net = tf.reshape(net, [-1, 6*6*64])
        # 第五层是全连接层，输出个数为384
        net = fc(net, 384, scope='fc3')
        # 第六层是全连接层，输出个数为192
        net = fc(net, 192, scope='fc4')
        # 第七层是全连接层，输出个数为10，注意这里不要使用激活函数
        net = fc(net, 10, scope='fc5', act=tf.identity)
        return net
```

3、结果

AlexNet在训练集和测试集分别达到了0.97和0.72的准确率

```
[train]: step 0 loss = 2.3719 acc = 0.1094 (0.0101 / batch)
[val]: step 0 loss = 2.3539 acc = 0.1562
[train]: step 1000 loss = 1.2158 acc = 0.6250 (0.0299 / batch)
[train]: step 2000 loss = 1.1322 acc = 0.6719 (0.0300 / batch)
[train]: step 3000 loss = 0.6507 acc = 0.7969 (0.0300 / batch)
[train]: step 4000 loss = 0.3704 acc = 0.8750 (0.0299 / batch)
[val]: step 4000 loss = 0.9157 acc = 0.7188
[train]: step 5000 loss = 0.2648 acc = 0.9062 (0.0300 / batch)
[train]: step 6000 loss = 0.4816 acc = 0.8438 (0.0299 / batch)
[train]: step 7000 loss = 0.2262 acc = 0.9062 (0.0300 / batch)
[train]: step 8000 loss = 0.2691 acc = 0.9375 (0.0301 / batch)
[val]: step 8000 loss = 1.1058 acc = 0.7812
[train]: step 9000 loss = 0.3104 acc = 0.9219 (0.0300 / batch)
[train]: step 10000 loss = 0.4133 acc = 0.8594 (0.0300 / batch)
[train]: step 11000 loss = 0.0668 acc = 0.9688 (0.0299 / batch)
[train]: step 12000 loss = 0.0874 acc = 0.9844 (0.0299 / batch)
[val]: step 12000 loss = 1.3469 acc = 0.7656
[train]: step 13000 loss = 0.2461 acc = 0.9375 (0.0300 / batch)
[train]: step 14000 loss = 0.0157 acc = 1.0000 (0.0300 / batch)
[train]: step 15000 loss = 0.5685 acc = 0.9688 (0.0300 / batch)
[train]: step 16000 loss = 0.1240 acc = 0.9375 (0.0300 / batch)
[val]: step 16000 loss = 2.1229 acc = 0.7031
[train]: step 17000 loss = 0.2062 acc = 0.9219 (0.0300 / batch)
[train]: step 18000 loss = 0.0268 acc = 0.9844 (0.0301 / batch)
[train]: step 19000 loss = 0.0972 acc = 0.9844 (0.0284 / batch)
[train]: step 20000 loss = 0.0555 acc = 0.9844 (0.0306 / batch)
[val]: step 20000 loss = 1.1873 acc = 0.7031
--------------------Over all Result--------------------
[TRAIN]: loss = 0.0892 acc = 0.9716
[VAL]: loss = 1.8936 acc = 0.7366
```

高层API-keras和TF-Slim的使用

1、keras

大部分时候人们都倾向于使用对tensorflow底层代码进行封装的**高层api**，比如**keras,slim,tflearn,skflow**等等.在这里,我们来分别使用**keras和slim**这两个非常流行的高层api尝试构造AlexNet

构建Keras模型

Keras网络层

Keras为了方便用户搭建神经网络模型, 把很多常用的层, 比如Conv2d, MaxPooling2d,封装起来, 使得输入更加简单明了.

Keras模型

Keras提供Sequential和Model两种模型的构建方法, 使用他们搭建模型就像搭积木一样非常直观简单.

```
model.add(Conv2D(64, (5, 5), input_shape=(32, 32, 3)))
model.add(Activation('relu'))
```

```
model.add(MaxPooling2D([3, 3], 2))
model.add(Conv2D(64, (5, 5), activation='relu'))
model.add(MaxPooling2D([3, 3], 2))
model.add(Flatten())
model.add(Dense(384, activation='relu'))
model.add(Dense(192, activation='relu'))
model.add(Dense(10, activation='softmax'))
```

模型编译

模型构建完成之后, 我们需要用compile来配置训练过程
model.compile()接受三个参数:

- **optimizer:** 优化方法, 有**“sgd”,”rmsprop”,”adgrad”**等这样的字符串, 也可以是keras.Optimizers对象
- **loss**: 损失函数, 有**categorical_crossentropy, mes**等这样的字符串, 也可以是函数形式
- **metrics**: 评价函数, 如**[‘accuracy’]** , 也支持自定义

```
sgd = optimizers.SGD(lr=0.01, momentum=0.9)
model.compile(optimizer=sgd, loss="categorical_crossentropy", metrics=['accuracy'])
```

训练

```
model.fit(x=x_train, y=onehot_train, epochs=25, batch_size=64)
```

2、Slim

slim中的高级层
slim也对tensorflow的底层API进行了层的封装, 像Keras一样, 它也具有

- slim.conv2d
- slim.max_pool2d
- slim.flatten
- slim.fully_connected
- slim.batch_norm

arg_scope

在构建模型的时候会遇到很多相同的参数, 比如说很多卷积层或者池化层的补零策略都是”VALID”或者”SAME”, 很多变量的初始化函数都是tf.truncated_normal_initializer或者tf.constant_initializer, 如果全部手动写就会显得非常麻烦.这个时候, 可以通过**python的with语句和slim.arg_scope()**构成一个**参数域**, 在这个域下一次性定义好所有函数的一些默认参数, 就会非常方便了

使用arg_scope分为**两个步骤:**

- 定义你要对哪些函数使用默认参数
- 定义你要使用的默认参数的具体值

```
# 定义`AlexNet`的默认参数域
def alexnet_arg_scope():
    # 首先我们定义卷积和全连接层的参数域，他们都用`tf.nn.relu`作为激活函数
    # 都用`tf.truncated_normal`作为权重的初始化函数
    with slim.arg_scope([slim.conv2d, slim.fully_connected],
                        activation_fn=tf.nn.relu,
                        weights_initializer=tf.truncated_normal_initialize
r(stddev=1e-2)):
        # 在参数域内部继续定义参数域，这里，我们注意到在`AlexNet`中
        # 卷积和池化的补零策略都是`VALID`.
        with slim.arg_scope([slim.conv2d, slim.max_pool2d],
                            padding='VALID') as sc:
            return sc
```

VGG

1、基本概念

VGG16相比AlexNet的一个改进是采用**连续的几个3x3的卷积核代替AlexNet中的较大卷积核（11x11，7x7，5x5）**。对于给定的感受野（与输出有关的输入图片的局部大小），**采用堆积的小卷积核是优于采用大的卷积核，因为多层非线性层可以增加网络深度来保证学习更复杂的模式，而且代价还比较小（参数更少）**。

vgg 的一个关键就是使用很**多层 3 x 3** 的卷积然后再使用一个**最大池化层**，这个模块被使用了很多次

2、代码实现

vgg是一个**不断堆叠的网络结构**,由**多个block**组成，我们先编写**单个block**:

```
def vgg_block(inputs, num_convs, out_depth, scope='vgg_block', reuse=Non
e):
    """构建vgg_block.
    一个 vgg_block 由`num_convs`个卷积层和一个最大值池化层构成.
    Args:
        inputs: 输入
        num_convs: 这一个block里卷积层的个数
        out_depth: 每一个卷积层的卷积核个数
        scope: 变量域名
        reuse: 是否复用
    """
    in_depth = inputs.get_shape().as_list()[-1]
    with tf.variable_scope(scope, reuse=reuse) as sc:
        net = inputs
        # 循环定义`num_convs`个卷积层
        for i in range(num_convs):
```

```
            net = conv(net, ksize=[3, 3], out_depth=out_depth, strides=
[1, 1], padding='SAME', scope='conv%d' % i, reuse=reuse)
        net = max_pool(net, [2, 2], [2, 2], name='pool')
        return net
```

然后把很多个不同的vgg_block堆叠在一起

```
def vgg_stack(inputs, num_convs, out_depths, scope='vgg_stack', reuse=Non
e):
    """构建vgg_stack.
    一个 vgg_stack 将若干个不同的`vgg_block`进行`stack`(堆叠)
    Args:
        inputs: 输入
        num_convs: 每一个block里卷积层的个数, 列表. 如`[1, 2, 3]`
        out_depths: 每一个block的卷积核个数, 列表, 如`[64, 128, 256]`
        scope: 变量域名
        reuse: 是否复用
    """
    with tf.variable_scope(scope, reuse=reuse) as sc:
        net = inputs
        for i, (n, d) in enumerate(zip(num_convs, out_depths)):    #将对象中
对应的元素打包成一个个元组，然后返回由这些元组组成的列表。
            net = vgg_block(net, n, d, scope='block%d' % i)
        return net
```

通过几个全连接层将vgg搭建完成

```
def vgg(inputs, num_convs, out_depths, num_outputs, scope='vgg', reuse=No
ne):
    """构建vgg.
    一个 vgg 先经过`vgg_stack`后再连接两个全连接层.
    Args:
        inputs: 输入
        num_convs: 每一个 vgg_block 的卷积层的个数
        out_depths: 每一个 vgg_block 卷积核个数
        num_outputs: 最后输出向量的维数
        scope: 变量域名
        reuse: 是否复用
    """
    with tf.variable_scope(scope, reuse=reuse) as sc:
        net = vgg_stack(inputs, num_convs, out_depths)
        with tf.variable_scope('classification'):
            net = tf.reshape(net, (batch_size, -1))
            net = fc(net, 100, scope='fc1')
            net = fc(net, num_outputs, act=tf.identity, scope='classifica
tion')
        return net
```

3、结果

`VGG` 在训练集和测试集分别达到了 `0.97` 和 `0.75` 的准确率

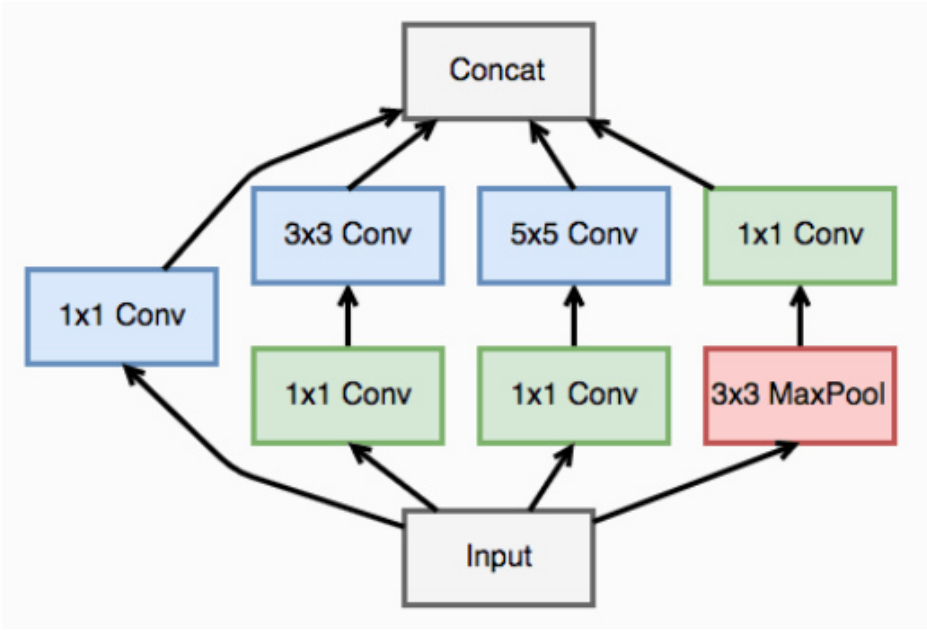
```
[train]: step 3000 loss = 0.9806 acc = 0.6875 (0.0408 / batch)
[train]: step 4000 loss = 0.7159 acc = 0.7344 (0.0407 / batch)
[val]: step 4000 loss = 0.7271 acc = 0.8125
[train]: step 5000 loss = 0.3444 acc = 0.8750 (0.0409 / batch)
[train]: step 6000 loss = 0.6834 acc = 0.8281 (0.0408 / batch)
[train]: step 7000 loss = 0.1895 acc = 0.9375 (0.0409 / batch)
[train]: step 8000 loss = 0.3990 acc = 0.8594 (0.0411 / batch)
[val]: step 8000 loss = 0.7746 acc = 0.8125
[train]: step 9000 loss = 0.3234 acc = 0.8906 (0.0411 / batch)
[train]: step 10000 loss = 0.2788 acc = 0.9219 (0.0409 / batch)
[train]: step 11000 loss = 0.2191 acc = 0.9219 (0.0409 / batch)
[train]: step 12000 loss = 0.0847 acc = 0.9844 (0.0406 / batch)
[val]: step 12000 loss = 1.2664 acc = 0.7344
[train]: step 13000 loss = 0.1677 acc = 0.9531 (0.0410 / batch)
[train]: step 14000 loss = 0.0701 acc = 0.9688 (0.0410 / batch)
[train]: step 15000 loss = 0.1077 acc = 0.9688 (0.0412 / batch)
[train]: step 16000 loss = 0.3446 acc = 0.9375 (0.0410 / batch)
[val]: step 16000 loss = 1.3316 acc = 0.7031
[train]: step 17000 loss = 0.0582 acc = 0.9844 (0.0405 / batch)
[train]: step 18000 loss = 0.1029 acc = 0.9844 (0.0408 / batch)
[train]: step 19000 loss = 0.2331 acc = 0.9375 (0.0407 / batch)
[train]: step 20000 loss = 0.0586 acc = 0.9844 (0.0409 / batch)
[val]: step 20000 loss = 1.7949 acc = 0.7188
————————————————————Over all Result————————————————————
[TRAIN]: loss = 0.0524 acc = 0.9849
[VAL]: loss = 1.2592 acc = 0.7644
```

GoogleNet

1、基本概念

InceptionNet采用了一种非常有效的 inception 模块，得到了比 VGG 更深的网络结构，但是却比 VGG 的**参数更少**，因为其去掉了**后面的全连接层**，所以参数大大减少，同时有了**很高的计算效率。**

- **1.深度**，层数更深，文章采用了22层，为了避免上述提到的**梯度消失问题**，googlenet巧妙的在不同深度处增加了**两个loss来保证梯度回传消失的现象。**
- **2.宽度**，增加了**多种核 1x1，3x3，5x5，还有直接max pooling的**，但是如果简单的将这些应用到feature map上的话，concat起来的feature map厚度将会很大，所以在googlenet中为了避免这一现象提出的inception具有如下结构，在3x3前，5x5前，max pooling后分别加上了**1x1的卷积核起到了降低feature map厚度**的作用。



一个 inception 模块的四个并行线路如下：

- 1.一个 1 x 1 的卷积，一个**小的感受野进行卷积提取特征**
- 2.一个 1 x 1 的卷积加上一个 3 x 3 的卷积，1 x 1 的卷积**降低输入的特征通道**，减少参数计算量，然后接一个 3 x 3 的卷积做一个**较大感受野的卷积**
- 3.一个 1 x 1 的卷积加上一个 5 x 5 的卷积，作用和第二个一样
- 4.一个 3 x 3 的最大池化加上 1 x 1 的卷积，最大池化**改变输入的特征排列**，1 x 1 的卷积进行特征提取

2、代码实现

构建一个inception模块

```
def inception(x, d0_1, d1_1, d1_3, d2_1, d2_5, d3_1, scope='inception', r
euse=None):
    with tf.variable_scope(scope, reuse=reuse):
        # 我们把`slim.conv2d`,`slim.max_pool2d`的默认参数放在`slim`的参数域里
        with slim.arg_scope([slim.conv2d, slim.max_pool2d], stride=1, pad
ding='SAME'):
            # 第一个分支
            with tf.variable_scope('branch0'):
                branch_0 = slim.conv2d(x, d0_1, [1, 1], scope='conv_1x1')
            # 第二个分支
            with tf.variable_scope('branch1'):
                branch_1 = slim.conv2d(x, d1_1, [1, 1], scope='conv_1x1')
                branch_1 = slim.conv2d(branch_1, d1_3, [3, 3], scope='con
v_3x3')
            # 第三个分支
            with tf.variable_scope('branch2'):
                branch_2 = slim.conv2d(x, d2_1, [1, 1], scope='conv_1x1')
                branch_2 = slim.conv2d(branch_2, d2_1, [5, 5], scope='con
v_5x5')
```

```
            # 第四个分支
            with tf.variable_scope('branch3'):
                branch_3 = slim.max_pool2d(x, [3, 3], scope='max_pool')
                branch_3 = slim.conv2d(branch_3, d3_1, [1, 1], scope='con
v_1x1')
            # 连接
            net = tf.concat([branch_0, branch_1, branch_2, branch_3], axi
s=-1)
            return net
```

使用单个inception模块去构建整个googlenet

```
def googlenet(inputs, num_classes, reuse=None, is_training=None, verbose=
False):
    with tf.variable_scope('googlenet', reuse=reuse):
        # 给`batch_norm`的`is_training`参数设定默认值.
        # `batch_norm`和`is_training`密切相关，当`is_trainig=True`时，
        # 它使用的是一个`batch`数据的移动平均,方差值
        # 当`is_training=True`时，它使用的是固定值
        with slim.arg_scope([slim.batch_norm], is_training=is_training):
            with slim.arg_scope([slim.conv2d, slim.max_pool2d, slim.avg_p
ool2d], padding='SAME', stride=1):
                net = inputs
                with tf.variable_scope('block1'):
                    net = slim.conv2d(net, 64, [5, 5], stride=2, scope='c
onv_5x5')
                    if verbose:
                        print('block1 output: {}'.format(net.shape))
                with tf.variable_scope('block2'):
                    net = slim.conv2d(net, 64, [1, 1], scope='conv_1x1')
                    net = slim.conv2d(net, 192, [3, 3], scope='conv_3x3')
                    net = slim.max_pool2d(net, [3, 3], stride=2, scope='m
ax_pool')
                    if verbose:
                        print('block2 output: {}'.format(net.shape))
                with tf.variable_scope('block3'):
                    net = inception(net, 64, 96, 128, 16, 32, 32, scope
='inception_1')
                    net = inception(net, 128, 128, 192, 32, 96, 64, scope
='inception_2')
                    net = slim.max_pool2d(net, [3, 3], stride=2, scope='m
ax_pool')
                    if verbose:
                        print('block3 output: {}'.format(net.shape))
                with tf.variable_scope('block4'):
                    net = inception(net, 192, 96, 208, 16, 48, 64, scope
='inception_1')
                    net = inception(net, 160, 112, 224, 24, 64, 64, scope
='inception_2')
                    net = inception(net, 128, 128, 256, 24, 64, 64, scope
='inception_3')
```

```
                    net = inception(net, 112, 144, 288, 24, 64, 64, scope
='inception_4')
                    net = inception(net, 256, 160, 320, 32, 128, 128, sco
pe='inception_5')
                    net = slim.max_pool2d(net, [3, 3], stride=2, scope='m
ax_pool')
                    if verbose:
                        print('block4 output: {}'.format(net.shape))
                with tf.variable_scope('block5'):
                    net = inception(net, 256, 160, 320, 32, 128, 128, sco
pe='inception1')
                    net = inception(net, 384, 182, 384, 48, 128, 128, sco
pe='inception2')
                    net = slim.avg_pool2d(net, [2, 2], stride=2, scope='a
vg_pool')
                    if verbose:
                        print('block5 output: {}'.format(net.shape))
                with tf.variable_scope('classification'):
                    net = slim.flatten(net)
                    net = slim.fully_connected(net, num_classes, activati
on_fn=None, normalizer_fn=None, scope='logit')
                    if verbose:
                        print('classification output: {}'.format(net.shap
e))
                return net
```

3、结果

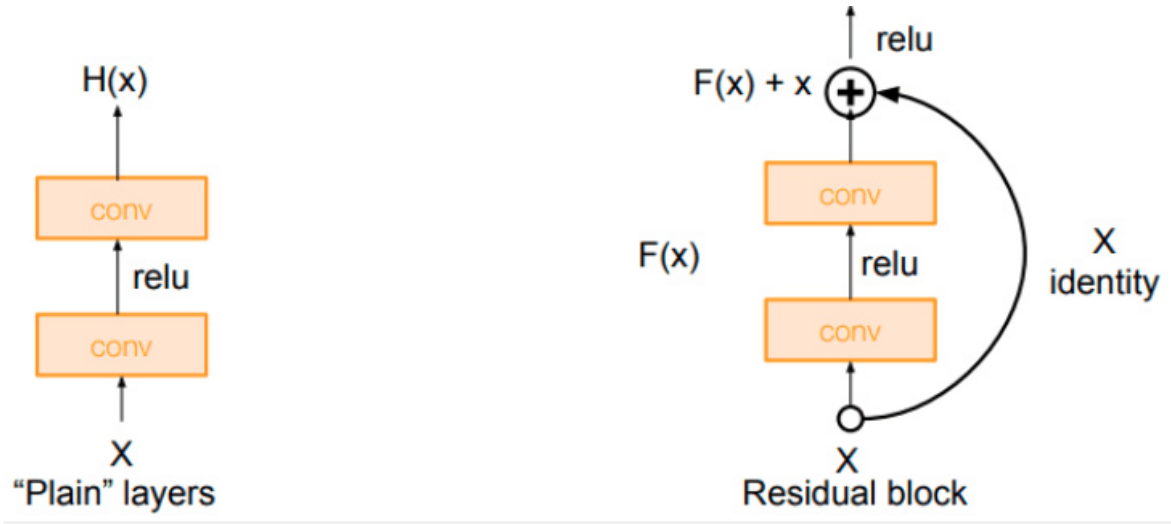
```
[train]: step 3000 loss = 0.6100 acc = 0.7500 (0.1268 / batch)
[train]: step 4000 loss = 0.6768 acc = 0.8125 (0.1269 / batch)
[val]: step 4000 loss = 0.9132 acc = 0.6719
[train]: step 5000 loss = 0.6148 acc = 0.8125 (0.1271 / batch)
[train]: step 6000 loss = 1.0494 acc = 0.6875 (0.1270 / batch)
[train]: step 7000 loss = 0.3237 acc = 0.8750 (0.1273 / batch)
[train]: step 8000 loss = 0.3010 acc = 0.8438 (0.1273 / batch)
[val]: step 8000 loss = 1.1506 acc = 0.7344
[train]: step 9000 loss = 0.2065 acc = 0.9531 (0.1274 / batch)
[train]: step 10000 loss = 0.2854 acc = 0.9062 (0.1271 / batch)
[train]: step 11000 loss = 0.2809 acc = 0.9062 (0.1272 / batch)
[train]: step 12000 loss = 0.2762 acc = 0.9062 (0.1267 / batch)
[val]: step 12000 loss = 1.8881 acc = 0.6562
[train]: step 13000 loss = 0.2256 acc = 0.9062 (0.1271 / batch)
[train]: step 14000 loss = 0.0982 acc = 0.9531 (0.1271 / batch)
[train]: step 15000 loss = 0.1602 acc = 0.9375 (0.1270 / batch)
[train]: step 16000 loss = 0.2146 acc = 0.9844 (0.1267 / batch)
[val]: step 16000 loss = 1.1134 acc = 0.8281
[train]: step 17000 loss = 0.0645 acc = 0.9688 (0.1272 / batch)
[train]: step 18000 loss = 0.0383 acc = 0.9844 (0.1249 / batch)
[train]: step 19000 loss = 0.0116 acc = 1.0000 (0.1246 / batch)
[train]: step 20000 loss = 0.1224 acc = 0.9688 (0.1248 / batch)
[val]: step 20000 loss = 1.0764 acc = 0.8125
————————————————————Over all Result————————————————————
[TRAIN]: loss = 0.0342 acc = 0.9886
[VAL]: loss = 1.2499 acc = 0.7835
```

ResNet

1、基本概念

ResNet 有效地解决了深度神经网络难以训练的问题，可以训练高达 1000 层的卷积网络。网络之所以难以训练，是因为存在着**梯度消失的问题**，离 **loss 函数越远的层，在反向传播的时候，梯度越小，就越难以更新**，随着层数的增加，这个现象越严重。之前有两种常见的方案来解决这个问题：

- 1.按层训练，先训练比较浅的层，然后在不断增加层数，但是这种方法效果不是特别好，而且比较麻烦
- 2.使用更宽的层，或者增加输出通道，而不加深网络的层数，这种结构往往得到的效果又不好



这就普通的网络连接跟跨层残差连接的对比图，使用普通的连接，上层的梯度必须要一层一层传回来，而是用残差连接，相当于中间有了一条更短的路，梯度能够**从这条更短的路传回来，避免了梯度过小**的情况。

假设某层的输入是 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，这就是一个更浅层的网络，更容易训练，而这个网络没有学会的部分，我们可以使用更深的网络 F(x) 去训练它，使得训练更加容易，最后希望拟合的结果就是 **F(x) = H(x) - x**，这就是一个残差的结构

2、代码实现

定义一个下采样函数

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

构建resnet整体结构

```
def resnet(inputs, num_classes, reuse=None, is_training=None, verbose=Fal
se):
    with tf.variable_scope('resnet', reuse=reuse):
        net = inputs
        if verbose:
            print('input: {}'.format(net.shape))
        with slim.arg_scope([slim.batch_norm], is_training=is_training):
            with slim.arg_scope([slim.conv2d, slim.max_pool2d, slim.avg_p
ool2d], padding='SAME'):
                with tf.variable_scope('block1'):
                    net = slim.conv2d(net, 32, [5, 5], stride=2, scope='c
onv_5x5')
                    if verbose:
                        print('block1: {}'.format(net.shape))
                                with tf.variable_scope('block2'):
```

```
                    net = slim.max_pool2d(net, [3, 3], 2, scope='max_poo
l')
                    net = residual_block(net, 32, 128, scope='residual_bl
ock1')
                    net = residual_block(net, 32, 128, scope='residual_bl
ock2')
                    if verbose:
                        print('block2: {}'.format(net.shape))
                with tf.variable_scope('block3'):
                    net = residual_block(net, 64, 256, stride=2, scope='r
esidual_block1')
                    net = residual_block(net, 64, 256, scope='residual_bl
ock2')
                    if verbose:
                        print('block3: {}'.format(net.shape))
                with tf.variable_scope('block4'):
                    net = residual_block(net, 128, 512, stride=2,
scope='residual_block1')
                    net = residual_block(net, 128, 512, scope='residual_b
lock2')
                    if verbose:
                        print('block4: {}'.format(net.shape))
                with tf.variable_scope('classification'):
                    net = tf.reduce_mean(net, [1, 2], name='global_pool',
 keep_dims=True)
                    net = slim.flatten(net, scope='flatten')
                    net = slim.fully_connected(net, num_classes, activati
on_fn=None, normalizer_fn=None, scope='logit')
                    if verbose:
                        print('classification: {}'.format(net.shape))
                return net
```

residual_block

```
def residual_block(x, bottleneck_depth, out_depth, stride=1, scope='resid
ual_block'):
    in_depth = x.get_shape().as_list()[-1]
    with tf.variable_scope(scope):
        # 如果通道数没有改变,用下采样改变输入的大小
        if in_depth == out_depth:
            shortcut = subsample(x, stride, 'shortcut')
        # 如果有变化，用卷积改变输入的通道以及大小
        else:
            shortcut = slim.conv2d(x, out_depth, [1, 1], stride=stride, a
ctivation_fn=None, scope='shortcut')
        residual = slim.conv2d(x, bottleneck_depth, [1, 1], stride=1, sco
pe='conv1')
        residual = slim.conv2d(residual, bottleneck_depth, 3, stride, sco
pe='conv2')
        residual = slim.conv2d(residual, out_depth, [1, 1], stride=1, act
ivation_fn=None, scope='conv3')
```

```
        # 相加操作
        output = tf.nn.relu(shortcut + residual)
        return output
```

3、结果

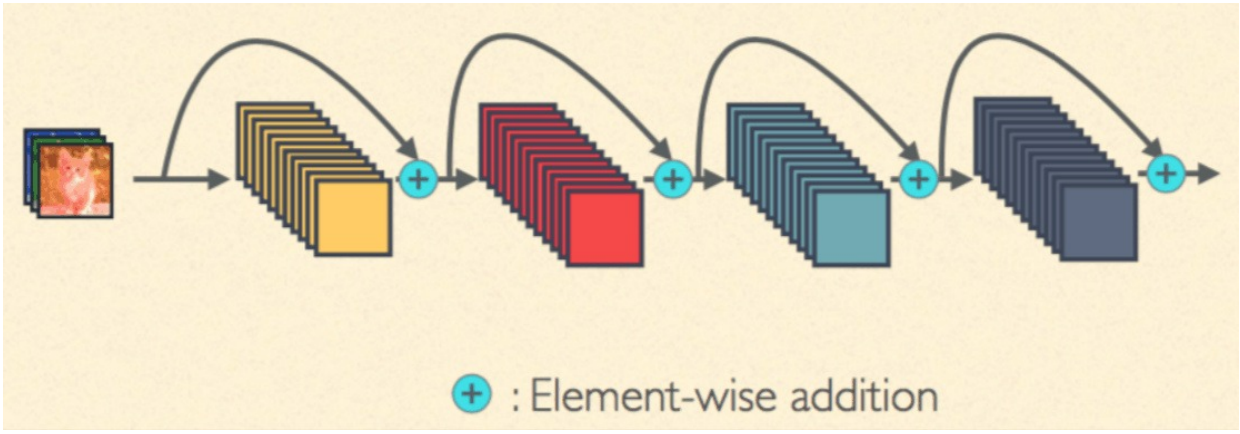
```
[val]: step 4000 loss = 1.2665 acc = 0.5938
[train]: step 5000 loss = 0.4954 acc = 0.8281 (0.1228 / batch)
[train]: step 6000 loss = 0.8432 acc = 0.7188 (0.1227 / batch)
[train]: step 7000 loss = 0.4143 acc = 0.8438 (0.1227 / batch)
[train]: step 8000 loss = 0.4730 acc = 0.8438 (0.1227 / batch)
[val]: step 8000 loss = 0.9876 acc = 0.7188
[train]: step 9000 loss = 0.2425 acc = 0.8906 (0.1226 / batch)
[train]: step 10000 loss = 0.2268 acc = 0.9375 (0.1225 / batch)
[train]: step 11000 loss = 0.6189 acc = 0.8125 (0.1226 / batch)
[train]: step 12000 loss = 0.3678 acc = 0.8750 (0.1224 / batch)
[val]: step 12000 loss = 1.9084 acc = 0.6719
[train]: step 13000 loss = 0.4811 acc = 0.8281 (0.1227 / batch)
[train]: step 14000 loss = 0.1818 acc = 0.9375 (0.1226 / batch)
[train]: step 15000 loss = 0.0622 acc = 0.9844 (0.1229 / batch)
[train]: step 16000 loss = 0.0944 acc = 0.9531 (0.1229 / batch)
[val]: step 16000 loss = 1.0909 acc = 0.8125
[train]: step 17000 loss = 0.0499 acc = 0.9844 (0.1235 / batch)
[train]: step 18000 loss = 0.3842 acc = 0.9219 (0.1234 / batch)
[train]: step 19000 loss = 0.0652 acc = 0.9688 (0.1231 / batch)
[train]: step 20000 loss = 0.1152 acc = 0.9688 (0.1235 / batch)
[val]: step 20000 loss = 2.2715 acc = 0.6094
--------------------Over all Result--------------------
[TRAIN]: loss = 0.1439 acc = 0.9498
[VAL]: loss = 1.5657 acc = 0.7276
```

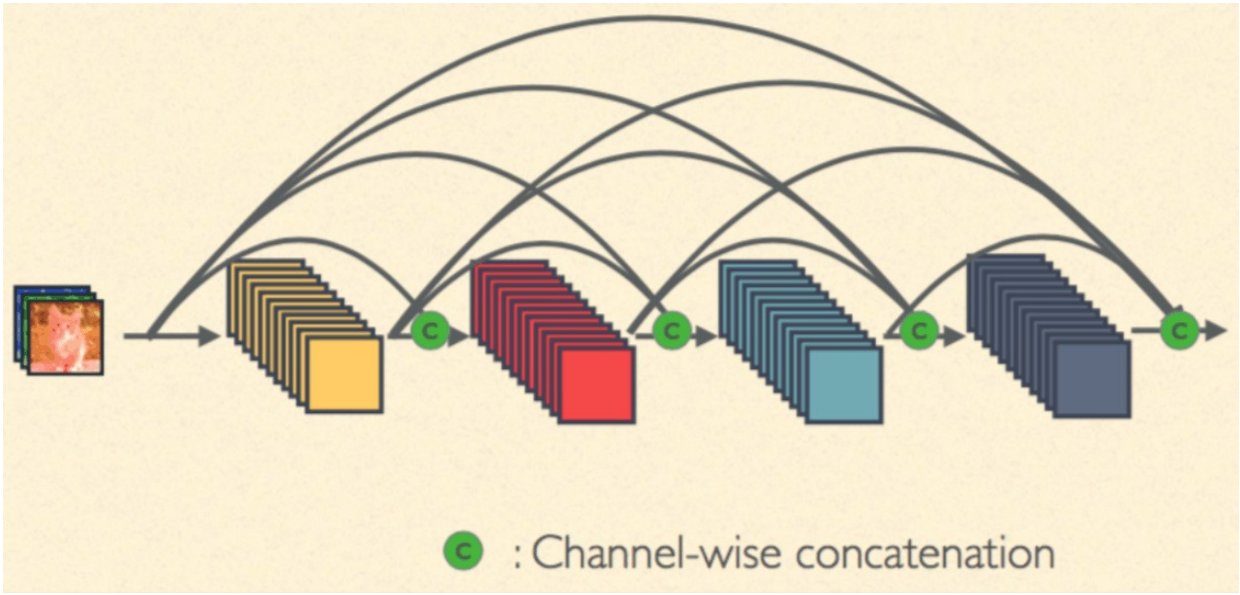
DenseNet

1、基本概念

DenseNet 主要由 **dense block** 构成,先列下DenseNet的**几个优点**，感受下它的强大：

- 1、减轻了vanishing-gradient（梯度消失）
- 2、加强了feature的传递
- 3、更有效地利用了feature
- 4、一定程度上较少了参数数量





第一张图是 ResNet，第二张图是 DenseNet，因为是在通道维度进行特征的拼接，所以**底层的输出会保留进入所有后面的层**，这能够更好的保证梯度的传播，同时能够**使用低维的特征和高维的特征进行联合训练**，能够得到更好的结果。

Bottleneck layer

每个dense block的3*3卷积前面都包含了一个**1*1的卷积操作**，目的是**减少输入的feature map数量，既能降维减少计算量**，又能融合各个通道的特征，何乐而不为。

Translation layer

在每**两个dense block**之间又增加了**1*1的卷积操作。**该层的1*1卷积的输出channel默认是输入channel到一半,为了**进一步压缩参数**.

2、代码实现

residual_block

```
def bn_relu_conv(x, out_depth, scope='dense_basic_conv', reuse=None):
    # 基本卷积单元是: bn->relu-conv
    with tf.variable_scope(scope, reuse=reuse):
        net = slim.batch_norm(x, activation_fn=None, scope='bn')
        net = tf.nn.relu(net, name='activation')
```

```
        net = slim.conv2d(net, out_depth, 3, activation_fn=None, normaliz
er_fn=None, biases_initializer=None, scope='conv')

        return net
```

构建densenet的基本单元

```
def dense_block(x, growth_rate, num_layers, scope='dense_block', reuse=No
ne):
    in_depth = x.get_shape().as_list()[-1]
    with tf.variable_scope(scope, reuse=reuse):
        net = x
        for i in range(num_layers):
            out = bn_relu_conv(net, growth_rate, scope='block%d' % i)
            # 将前面所有的输出连接到一起作为下一个基本卷积单元的输入
            net = tf.concat([net, out], axis=-1)
        return net
```

构建transition层

```
def transition(x, out_depth, scope='transition', reuse=None):
    in_depth = x.get_shape().as_list()[-1]
    with tf.variable_scope(scope, reuse=reuse):
        net = slim.batch_norm(x, activation_fn=None, scope='bn')
        net = tf.nn.relu(net, name='activation')
        net = slim.conv2d(net, out_depth, 1, activation_fn=None, normaliz
er_fn=None, biases_initializer=None, scope='conv')
        net = slim.avg_pool2d(net, 2, 2, scope='avg_pool')
        return net
```

3、结果

```
[train]: step 9000 loss = 0.0447 acc = 1.0000 (0.2206 / batch)
[train]: step 10000 loss = 0.0236 acc = 1.0000 (0.2205 / batch)
[train]: step 11000 loss = 0.0906 acc = 0.9531 (0.2207 / batch)
[train]: step 12000 loss = 0.0107 acc = 1.0000 (0.2208 / batch)
[val]: step 12000 loss = 1.3204 acc = 0.7500
[train]: step 13000 loss = 0.0167 acc = 1.0000 (0.2206 / batch)
[train]: step 14000 loss = 0.0025 acc = 1.0000 (0.2208 / batch)
[train]: step 15000 loss = 0.0003 acc = 1.0000 (0.2207 / batch)
[train]: step 16000 loss = 0.0041 acc = 1.0000 (0.2207 / batch)
[val]: step 16000 loss = 0.9162 acc = 0.8594
[train]: step 17000 loss = 0.0005 acc = 1.0000 (0.2207 / batch)
[train]: step 18000 loss = 0.0001 acc = 1.0000 (0.2206 / batch)
[train]: step 19000 loss = 0.0000 acc = 1.0000 (0.2207 / batch)
[train]: step 20000 loss = 0.0001 acc = 1.0000 (0.2206 / batch)
[val]: step 20000 loss = 0.7763 acc = 0.8750
----------------------Over all Result----------------------
[TRAIN]: loss = 0.0000 acc = 1.0000
[VAL]: loss = 0.7788 acc = 0.8498
```

训练技巧

1、数据增强

常用的数据增强方法如下：

- 1.对图片进行一定**比例缩放**
- 2.对图片进行**随机位置的截取**
- 3.对图片进行**随机的水平和竖直翻转**
- 4.对图片进行**随机角度的旋转**
- 5.对图片进行**亮度、对比度和颜色的随机变化**

原图

比例缩放

tensorflow内置函数**tf.image.resize_images**可以处理图片的放缩.

- 第一个参数是输入图片,
- 第二个参数是目标大小, 格式是[height, width].
- 第三个参数是resize时使用的方法, 默认是双先行插值, 可以在tf.image.ResizeMethod中选择

```
resized_im = tf.image.resize_images(im,
[100,200],method=tf.image.ResizeMethod.BILINEAR)
```

截取

随机位置截取能够提取出图片中局部的信息，使得网络接受的输入具有多尺度的特征，所以能够有较好的效果.

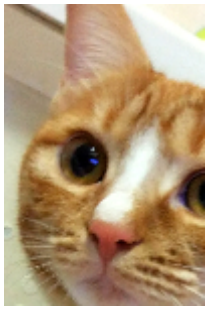
我们用tf.random_crop来实现.

- 第一个参数是输入图片,
- 第二个参数是截取区域的大小, 格式是[height, width, channel],
- 第三个参数是随机种子.

还可以用tf.image.central_crop来实现中心区域裁剪.

- 第一个参数是输入图片,
- 第二个参数是截取区域占原图比例, 也就是说默认是长宽同比例裁剪的

```
random_cropped_im2 = tf.random_crop(im, [150, 100, 3])
```



```
central_cropped_im = tf.image.central_crop(im, 1. / 3)
```

随机的水平和竖直方向翻转

对于上面这一张猫的图片，如果我们将它翻转一下，它仍然是一张猫，但是图片就有了更多的多样性，所以随机翻转也是一种非常有效的手段。在tensorflow中，随机翻转使用的是

tf.image.random_flip_up_and_down
tf.image.random_flip_left_and_right

亮度、对比度和颜色的变化

除了形状变化外，颜色变化又是另外一种增强方式，其中可以设置亮度变化，对比度变化和颜色变化等，在`tensorflow`中主要使用

- `tf.image.random_brightness`
- `tf.image.random_contrast`
- `tf.image.random_hue`

把所有的图像增强操作放在一起

```
def train_aug(im, scope='train_aug'):
    with tf.variable_scope(scope):
        im = tf.image.resize_images(im, [120, 120])
        im = tf.image.random_flip_left_right(im)
        im = tf.random_crop(im, [96, 96, 3])
        im = tf.image.random_brightness(im, max_delta=0.5)
        im = tf.image.random_contrast(im, lower=0.0, upper=0.5)
        im = tf.image.random_hue(im, max_delta=0.5)
```

```
            im = tf.image.per_image_standardization(im)
            return im
    def test_aug(im, scope='test_aug'):
        with tf.variable_scope(scope):
            im = tf.image.resize_images(im, [96, 96])
            im = tf.image.per_image_standardization(im)
            return im
```

这里需要对一个**batch**中所有图片进行增强, 我们需要用到**tf.map_fn**函数, 这个函数和python的map函数功能非常类似, 都能够对一个类似列表的数据结构进行函数操作, 而且比较快

```
train_imgs_aug = tf.map_fn(lambda image: train_aug(image), train_imgs)
val_imgs_aug = tf.map_fn(lambda image: test_aug(image), val_imgs)
```

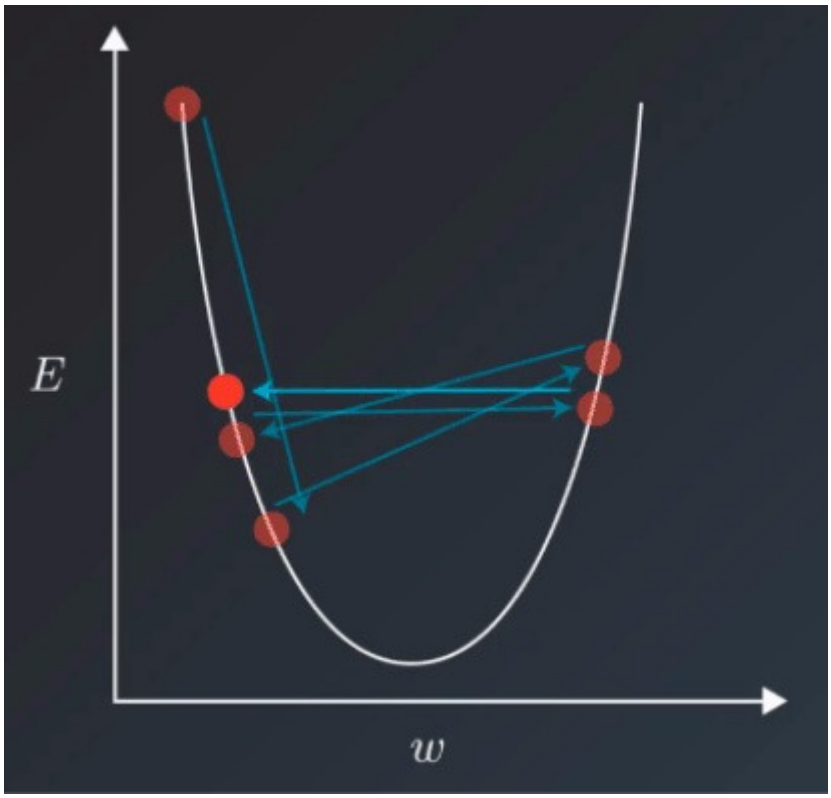
结果

对于训练集，不做数据增强跑 10000 次，准确率已经到了 82%，而使用了数据增强，跑 10 次准确率只有 68%，说明数据增强之后变得更难了。

而对于测试集，使用数据增强进行训练的时候，准确率会比不使用更高，因为数据增强**提高了模型应对于更多的不同数据集的泛化能力**，所以有更好的效果。

2、学习率衰减

对于基于一阶梯度进行优化的方法而言，开始的时候更新的幅度是比较大的，也就是说开始的学习率可以设置大一点，但是当**训练集的 loss 下降到一定程度之后，，使用这个太大的学习率就会导致 loss 一直来回震荡**，比如



在tensorflow中学习率衰减非常方便，使用 **tf.train.exponential_decay,** 但是它只支持指数式衰减和固定步长衰减

在这里, 我们用函数设计一个下降策略:

- 当训练步数小于12000时, 输出0.01
- 大于12000时输出0.001

```
def lr_step(step, **kwargs):
    lr = tf.cond(tf.less(step, 12000), lambda: 0.1, lambda: 0.01)
    return lr
```

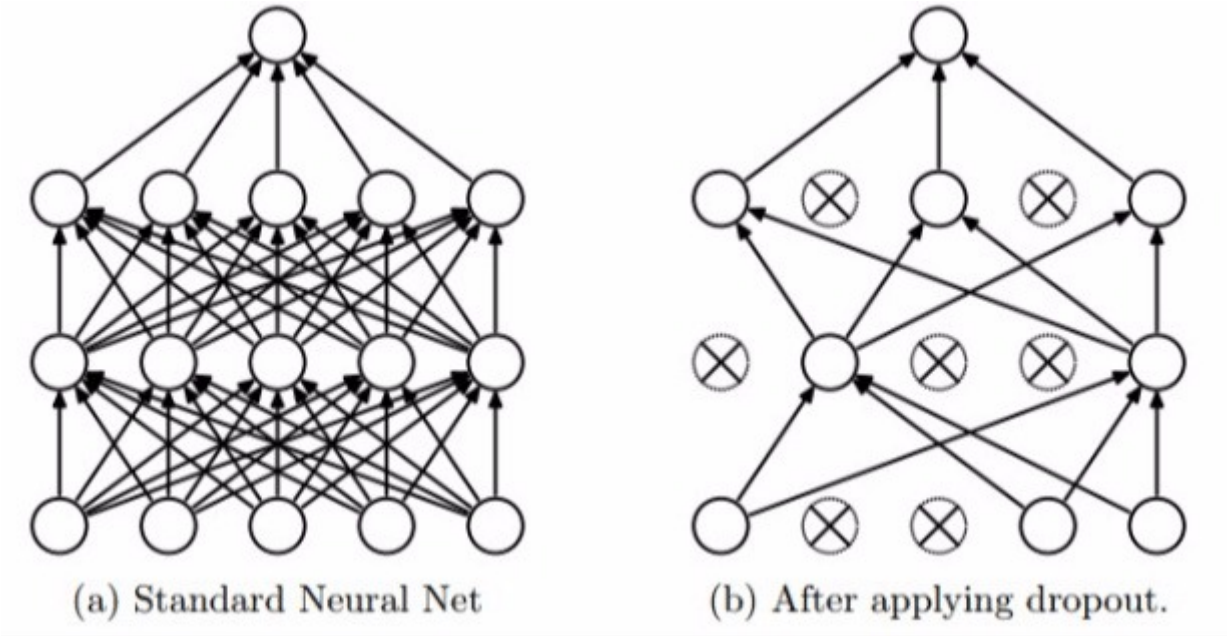
在这里我们用一个占位符来表示学习率, 方便在训练过程中我们从外部修改它

```
train_step = tf.Variable(0, trainable=False, name='train_step')
lr = lr_step(train_step)
opt = tf.train.MomentumOptimizer(lr, momentum=0.9)
```

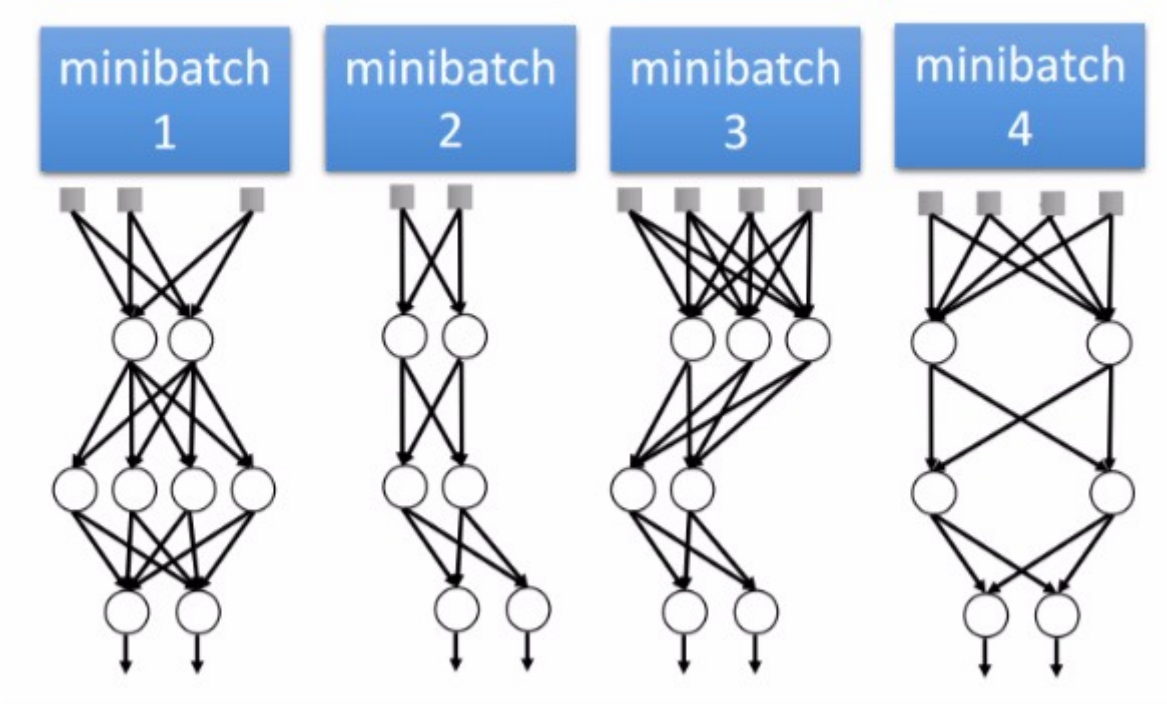
3、dropout

除了数据增强之外，改善过拟合的办法还有 **dropout。**

dropout 原理:在训练的时候以概率 p 保留每个神经元，也就是说在训练的时候，每次都会有神经元被随机设置为 0，如下图



左边是标准的神经网络，稠密连接，而右边就是使用了 dropout 的稀疏连接。我们可以看到每次训练的时候就会有某些神经元没有参与训练，所以在**每个 batch 进行训练的时候模型都会有微小的区别**，比如



使用 dropout 之前的输入是 x，那么使用完 dropout 之后输出的期望就是 $px+(1-p)0=px$，也就是说 $x \to px$。为了保证结果相同，非常简单，对输出做一下缩放，乘上 $\frac{1}{p}$ 就可以了。

`tensorflow` 中使用 `dropout` 的方法非常简答，使用 `tf.nn.dropuout` 就行了，第一个参数是输入, 第二个参数 `keep_prob` 表示保留的概率。对于 dropout 在训练和测试时候的表现不同，只需要将预测时的 `dropout` 改成 `1.0` 即可.

4、正则化

正则化是机器学习中提出来的一种方法，有 L1 和 L2 正则化，目前使用较多的是 L2 正则化，引入正则化相当于在 **loss 函数上面加上一项**，比如

$$f = loss + \lambda \sum_{p \in params} \|p\|_2^2$$

就是在 loss 的基础上加上了参数的二范数作为一个正则化，我们在训练网络的时候，不仅要最小化 loss 函数，同时还要最小化参数的二范数，也就是说**我们会对参数做一些限制，不让它变得太大。**

如果我们对新的损失函数 f 求导进行梯度下降，就有

$$\frac{\partial f}{\partial p_j} = \frac{\partial loss}{\partial p_j} + 2\lambda p_j$$

那么在更新参数的时候就有

$$p_j \to p_j - \eta(\frac{\partial loss}{\partial p_j} + 2\lambda p_j) = p_j - \eta \frac{\partial loss}{\partial p_j} - 2\eta\lambda p_j$$

可以看到 $p_j - \eta \frac{\partial loss}{\partial p_j}$ 和没加正则项要更新的部分一样，而后面的 $2\eta\lambda p_j$ 就是正则项的影响，可以看到加完正则项之后会对参数做更大程度的更新，这也被称为**权重衰减(weight decay)**，在 `tf-slim` 中正则项可以通过 `slim.arg_scope` 和 `slim.regularizers` 来实现，因为卷积层,全连接层都具有参数 `weight_regularzier`，因此我们使用 `slim.arg_scope([slim.conv2d, slim.fully_connected], weight_regularzier=slim.regularizers.l2_regularizer(weight_decay=0.0001))` 就可以实现所有卷积层的权重 `L2` 模衰减

注意正则项的系数的大小非常重要，如果**太大**，会**极大的抑制参数的更新**，导致欠拟合，如果**太小**，那么正则项这个部分**基本没有贡献**，所以选择一个合适的权重衰减系数非常重要，这个需要根据具体的情况去尝试，初步尝试可以使用 `1e-4` 或者 `1e-3`

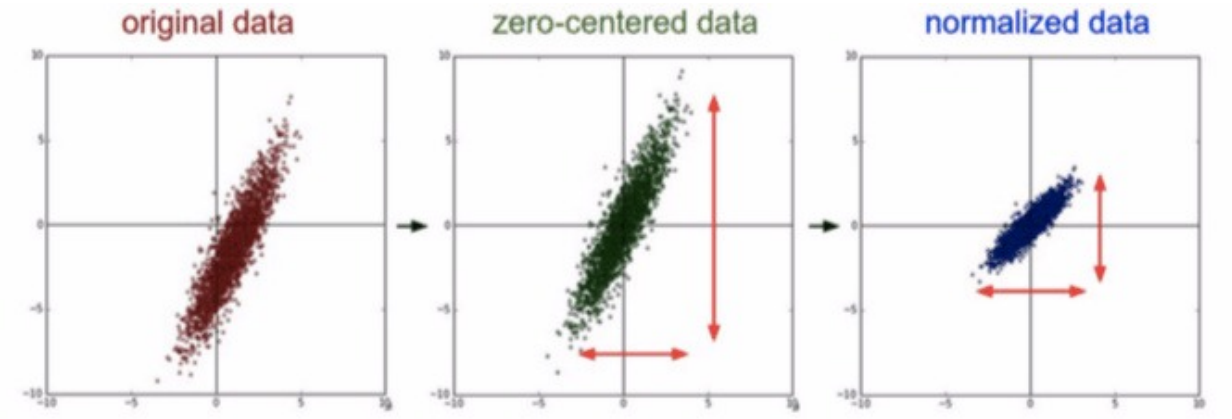
给所有的slim.conv2d和slim.fully_connected添加默认权重衰减, 用slim.arg_scope统一定义

```
with slim.arg_scope([slim.conv2d],activation_fn=tf.nn.relu,normalizer_fn
 = slim.batch_norm):
    with slim.arg_scope([slim.conv2d, slim.fully_connected], weights_regu
larizer = slim.regularizers.l2_regularizer(1e-4)) as sc:
                        conv_scope = sc
```

5、批量标准化

数据预处理

目前数据预处理最常见的方法就是**中心化和标准化**，中心化相当于修正数据的中心位置，实现方法非常简单，就是在每个特征维度上减去对应的均值，最后得到 0 均值的特征。标准化也非常简单，在数据变成 0 均值之后，为了使得不同的特征维度有着相同的规模，可以除以标准差近似为一个标准正态分布，也可以依据最大值和最小值将其转化为 -1 ~ 1 之间，下面是一个简单的图示



Batch Normalization

前面在数据预处理的时候，我们尽量输入特征不相关且满足一个标准的正态分布，这样模型的表现一般也较好。但是对于很深的网路结构，网路的非线性层会使得输出的结果**变得相关**，且不

再满足一个**标准的 N(0, 1) 的分布**，甚至输出的中心已经发生了偏移，这对于模型的训练，特别是深层的模型训练非常的困难。

批标准化，简而言之，就是**对于每一层网络的输出，对其做一个归一化，使其服从标准的正态分布**，这样后一层网络的输入也是一个标准的正态分布，所以能够比较好的进行训练，加快收敛速度。

batch normalization 的实现非常简单，对于给定的一个 batch 的数据 $B = \{x_1, x_2, \cdots, x_m\}$ 算法的公式如下

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$$

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$y_i = \gamma \hat{x}_i + \beta$$

第一行和第二行是计算出一个 batch 中数据的**均值和方差**，接着使用第三个公式对 batch 中的每个数据点做**标准化**，ϵ 是为了计算稳定引入的一个小的常数，通常取 10^{-5}，最后利用权重修正得到最后的输出结果，非常的简单

代码实现

使用批标准化在训练的时候能够很快地收敛

`tensorflow.contrib` 中内置了批标准化的函数 `tf.contrib.layers.batch_norm`，`tf-slim` 下有 `slim.batch_norm`，它们的函数接口也非常简单.

同时, 卷积层 `slim.conv2d` 具有输出标准化函数的参数 `normalizer_fn=None`, 默认是没有, 我们可以用 `slim.arg_scope` 来给每个卷积层附加一个批标准化函数.

```
def conv_bn_net(inputs, is_training, scope='conv_bn_net', reuse=None):
    with tf.variable_scope(scope, reuse=reuse):
        with slim.arg_scope([slim.conv2d], activation_fn=None, normalizer
_fn=slim.batch_norm):
            with slim.arg_scope([slim.batch_norm], is_training=is_trainin
g):
                net = slim.conv2d(inputs, 6, 3, scope='conv1')
                net = tf.nn.relu(net, name='activation1')
                net = slim.max_pool2d(net, 2, stride=2, scope='max_pool
1')
                net = slim.conv2d(net, 16, 5, scope='conv2')
                net = tf.nn.relu(net, name='activation2')
                net = slim.max_pool2d(net, 2, stride=2, scope='max_pool
2')
                net = slim.flatten(net, scope='flatten')
```

```
                net = slim.fully_connected(net, 10, activation_fn=None, s
cope='classification')
                return net
```

反馈与建议

- 微博：@柏林designer
- 邮箱：wwj123@zju.edu.cn