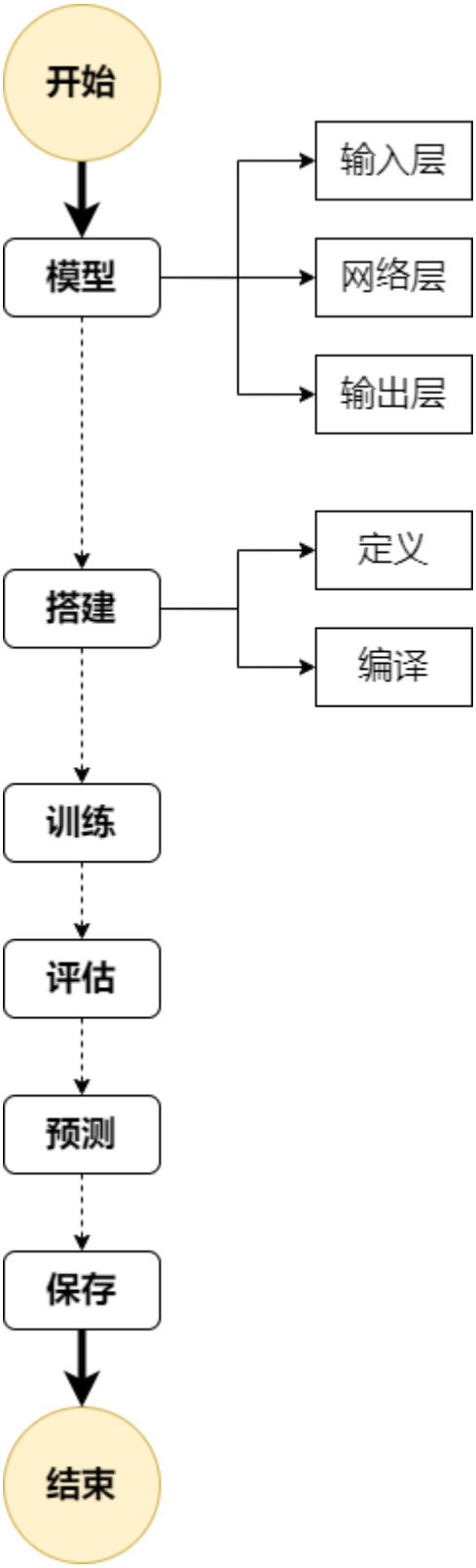
暑假学习记录

8.16(使用keras框架构建模型 and 残差网络的构建)

keras框架构建模型



1. 模型
    1. 输入层 输入层的作用就是规定了模型输入的shape。

```
from keras.layers import Input
ipt = Input(shape=(feature,))#shape不包含样本数量的维度
```

Input Input() 用于实例化 Keras 张量。

2. 网络层 网络层定义如：全连接层、卷积层、LSTM层、自定义网络层等。
一些常见网络层的主要参数如下：

Conv2D

keras.layers.Conv2D(filters, kernel_size, strides=(1, 1), padding='valid', data_format=None, dilation_rate=(1, 1), activation=None, use_bias=True, kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, bias_constraint=None)
2D 卷积层 (例如对图像的空间卷积)。
该层创建了一个卷积核， 该卷积核对层输入进行卷积， 以生成输出张量。 如果 use_bias 为 True， 则会创建一个偏置向量并将其添加到输出中。 最后，如果 activation 不是 None，它也会应用于输出。
当使用该层作为模型第一层时，需要提供 input_shape 参数 （整数元组，不包含样本表示的轴），例如， input_shape=(128, 128, 3) 表示 128x128 RGB 图像， 在 data_format="channels_last" 时。

Dense

keras.layers.Dense(units, activation=None, use_bias=True, kernel_initializer='glorot_uniform', bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, bias_constraint=None)
Dense 实现以下操作： output = activation(dot(input, kernel) + bias) 其中 activation 是按逐个元素计算的激活函数，kernel 是由网络层创建的权值矩阵，以及 bias 是其创建的偏置向量 (只在 use_bias 为 True 时才有用)。

Activation

将激活函数应用于输出。

Maxpooling2D

keras.layers.MaxPooling2D(pool_size=(2, 2), strides=None, padding='valid', data_format=None)
对于空间数据的最大池化。

BatchNormalization

keras.layers.BatchNormalization(axis=-1, momentum=0.99, epsilon=0.001, center=True, scale=True, beta_initializer='zeros', gamma_initializer='ones', moving_mean_initializer='zeros', moving_variance_initializer='ones', beta_regularizer=None, gamma_regularizer=None, beta_constraint=None, gamma_constraint=None)
批量标准化层 (Ioffe and Szegedy, 2014)。
在每一个批次的数据中标准化前一层的激活项， 即，应用一个维持激活项平均值接近 0，标准差接近 1 的转换。

Flatten

keras.layers.Flatten(data_format=None)
将输入展平。不影响批量大小。

3. 输出层 在Keras中没有像输入层一样专门的层定义，而是在模型的最后接入一个全连接层（Dense）作为输出层。

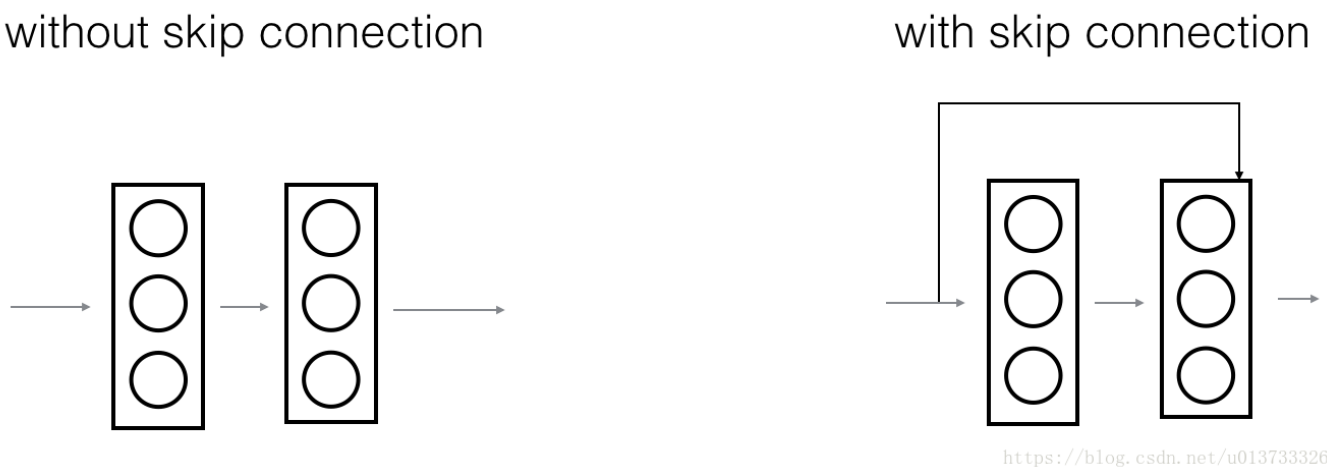
2. 搭建
   上一步，主要是定义一些层的基本信息，如输入输出、网络层（神经元数量、激活函数等等）。
   现在需要把这些层组建起来。
   1. 定义

```
model = Model(ipt,opt)     #定义模型，传入输入输出相应的层
model.summary()            #查看这个模型的网络层结构信息
```

   2. 编译
      编译，指明模型训练时用到的优化器、损失函数和评价标准等。
      compile(optimizer, loss=None, metrics=None, loss_weights=None,
      sample_weight_mode=None, weighted_metrics=None, target_tensors=None)
      用于配置训练模型。
      常见优化器：rmsprop、sgd adam
      还有损失函数和评价函数，评价函数可能并没有听过，其实只是用来评估好坏，并不作为返回值
   3. 训练
      fit(x=None, y=None, batch_size=None, epochs=1, verbose=1, callbacks=None,
      validation_split=0.0, validation_data=None, shuffle=True, class_weight=None,
      sample_weight=None, initial_epoch=0, steps_per_epoch=None, validation_steps=None)
      以给定数量的轮次（数据集上的迭代）训练模型。
   4. 评估
      evaluate(x=None, y=None, batch_size=None, verbose=1, sample_weight=None, steps=None)
      在测试模式下返回模型的误差值和评估标准值。
      计算是分批进行的。
   5. 预测
      predict(x, batch_size=None, verbose=0, steps=None)
      为输入样本生成输出预测。
      计算是分批进行的

残差网络的构建

最近几年，卷积神经网络变得越来越深，从只从几层（例如AlexNet）到超过一百层。使用深层网络最大的好处就是它能够完成很复杂的功能，它能够从边缘（浅层）到非常复杂的特征（深层）中不同的抽象层次的特征中学习。然而，使用比较深的网络通常没有什么好处，一个特别大的麻烦就在于训练的时候会产生梯度消失，非常深的网络通常会有一个梯度信号，该信号会迅速的消退，从而使得梯度下降变得非常缓慢。更具体的说，在梯度下降的过程中，当你从最后一层回到第一层的时候，你在每个步骤上乘以权重矩阵，因此梯度值可以迅速的指数式地减少到0（在极少数的情况下会迅速增长，造成梯度爆炸)。
在残差网络中，一个“捷径（shortcut）”或者说“跳跃连接（skip connection）”允许梯度直接反向传播到更浅的层，如下图：



图像左边是神经网络的主路，图像右边是添加了一条捷径的主路，通过这些残差块堆叠在一起，可以形成一个非常深的网络。
残差块有两种类型，主要取决于输入输出的维度是否相同。

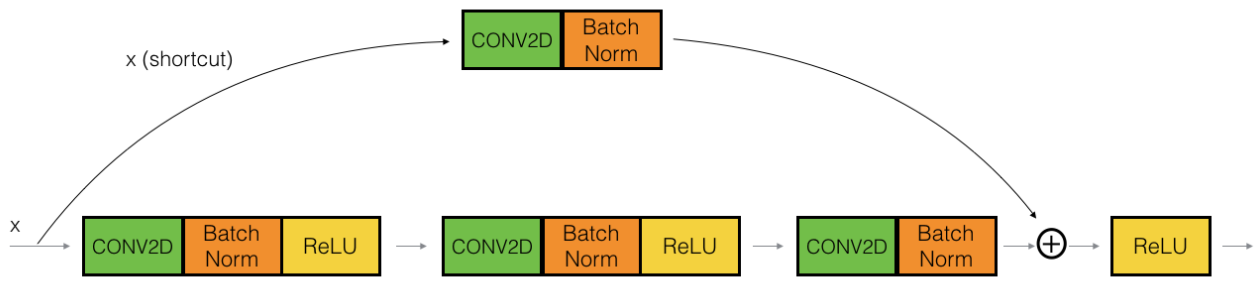
1. 恒等快
   恒等块是残差网络使用的的标准块，对应于输入的激活值（比如a[l]）与输出激活值（比如 a[l+1]）具有相同的维度。
   如下图：
   ！ 代码如下：

```
def identity_block(X, f, filters, stage, block):
"""
实现图3的恒等块

参数：
    X - 输入的tensor类型的数据，维度为( m, n_H_prev, n_W_prev, n_H_prev )
    f - 整数，指定主路径中间的CONV窗口的维度
    filters - 整数列表，定义了主路径每层的卷积层的过滤器数量
    stage - 整数，根据每层的位置来命名每一层，与block参数一起使用。
    block - 字符串，据每层的位置来命名每一层，与stage参数一起使用。

返回：
    X - 恒等块的输出，tensor类型，维度为(n_H, n_W, n_C)

"""

#定义命名规则
conv_name_base = "res" + str(stage) + block + "_branch"
bn_name_base   = "bn"  + str(stage) + block + "_branch"

#获取过滤器
F1, F2, F3 = filters

#保存输入数据，将会用于为主路径添加捷径
X_shortcut = X

#主路径的第一部分
##卷积层
```

```
X = Conv2D(filters=F1, kernel_size=(1,1), strides=(1,1)
,padding="valid",
           name=conv_name_base+"2a",
kernel_initializer=glorot_uniform(seed=0))(X)
##归一化
X = BatchNormalization(axis=3,name=bn_name_base+"2a")(X)
##使用ReLU激活函数
X = Activation("relu")(X)

#主路径的第二部分
##卷积层
X = Conv2D(filters=F2, kernel_size=(f,f),strides=(1,1),
padding="same",
           name=conv_name_base+"2b",
kernel_initializer=glorot_uniform(seed=0))(X)
##归一化
X = BatchNormalization(axis=3,name=bn_name_base+"2b")(X)
##使用ReLU激活函数
X = Activation("relu")(X)


#主路径的第三部分
##卷积层
X = Conv2D(filters=F3, kernel_size=(1,1), strides=(1,1),
padding="valid",
           name=conv_name_base+"2c",
kernel_initializer=glorot_uniform(seed=0))(X)
##归一化
X = BatchNormalization(axis=3,name=bn_name_base+"2c")(X)
##没有ReLU激活函数

#最后一步：
##将捷径与输入加在一起
X = Add()([X,X_shortcut])
##使用ReLU激活函数
X = Activation("relu")(X)

return X
```

2. 卷积块

我们已经实现了残差网络的恒等块，现在，残差网络的卷积块是另一种类型的残差块，它适用于输入输出的维度不一致的情况，它不同于上面的恒等块，与之区别在于，捷径中有一个CONV2D层，如下图：

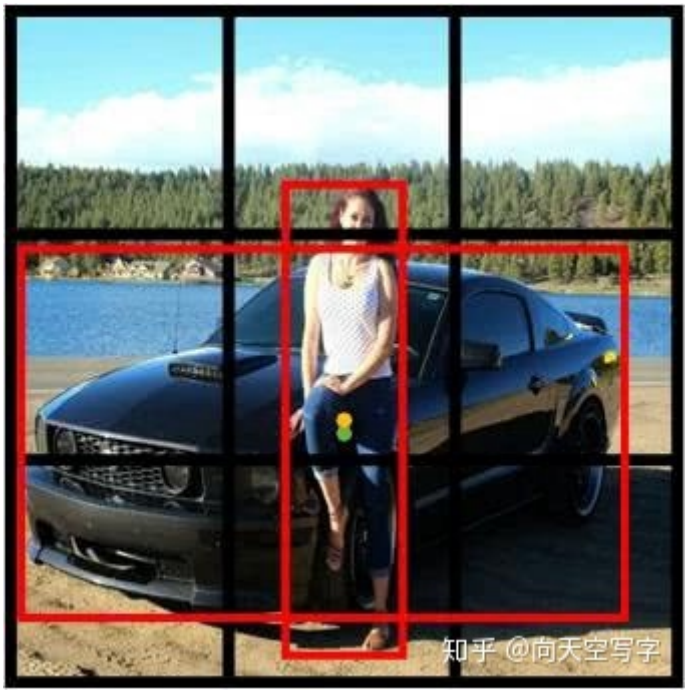


```
def convolutional_block(X, f, filters, stage, block, s=2):
"""
实现图5的卷积块

参数：
    X - 输入的tensor类型的变量，维度为( m, n_H_prev, n_W_prev, n_C_prev)
    f - 整数，指定主路径中间的CONV窗口的维度
    filters - 整数列表，定义了主路径每层的卷积层的过滤器数量
    stage - 整数，根据每层的位置来命名每一层，与block参数一起使用。
    block - 字符串，据每层的位置来命名每一层，与stage参数一起使用。
    s - 整数，指定要使用的步幅

返回：
    X - 卷积块的输出，tensor类型，维度为(n_H, n_W, n_C)
"""

#定义命名规则
conv_name_base = "res" + str(stage) + block + "_branch"
bn_name_base   = "bn"  + str(stage) + block + "_branch"

#获取过滤器数量
F1, F2, F3 = filters

#保存输入数据
X_shortcut = X

#主路径
##主路径第一部分
X = Conv2D(filters=F1, kernel_size=(1,1), strides=(s,s),
padding="valid",
           name=conv_name_base+"2a",
kernel_initializer=glorot_uniform(seed=0))(X)
X = BatchNormalization(axis=3,name=bn_name_base+"2a")(X)
X = Activation("relu")(X)

##主路径第二部分
X = Conv2D(filters=F2, kernel_size=(f,f), strides=(1,1),
padding="same",
           name=conv_name_base+"2b",
kernel_initializer=glorot_uniform(seed=0))(X)
X = BatchNormalization(axis=3,name=bn_name_base+"2b")(X)
X = Activation("relu")(X)

##主路径第三部分
X = Conv2D(filters=F3, kernel_size=(1,1), strides=(1,1),
padding="valid",
           name=conv_name_base+"2c",
kernel_initializer=glorot_uniform(seed=0))(X)
X = BatchNormalization(axis=3,name=bn_name_base+"2c")(X)

#捷径
X_shortcut = Conv2D(filters=F3, kernel_size=(1,1), strides=(s,s),
```

```
padding="valid",
           name=conv_name_base+"1",
kernel_initializer=glorot_uniform(seed=0))(X_shortcut)
X_shortcut = BatchNormalization(axis=3,name=bn_name_base+"1")
(X_shortcut)

#最后一步
X = Add()([X,X_shortcut])
X = Activation("relu")(X)

return X
```

8.18(YOLO算法 及实现车辆识别)

YOLO是You Only Look Once的缩写。这也是为了特别突出YOLO区别于两阶段算法的特点，从名字就可以感受到，YOLO算法速度很快，事实上也是如此。

1. Grid cell
   在YOLO中，目标图片被划分为Grid Cell，实际应用中，会使用19*19的grid cell。我们这里都先讨论一个格子里面只有一个目标，在训练集里，我们根据物体的中点将物体分配到一个格子(尽管物体的其他部分可能出现在其他格子里，但我们仍然认为其他格子里没有物体，物体由它中心所在的格子来负责预测。)。我们的输出是pc，bx,bh,bw,bh，以及C(预测的目标的概率)。
2. Anchor Box
   Anchor Box使得YOLO可以检测出同一个grid cell中包含多个物体的情况。 我们可能遇到两个物体的中心都在一个格子中，例如下图： 

上图中，人和车的中心都处于中间的grid cell中。Anchor box为标签增加了更多的纬度。如果我们可以对每个物体对应一个anchor box来标识。为了解释方便，这里我们只使用两个anchor box。每一个grid cell包含两个anchor box，意味着每一个grid cell可以预测两个物体。至于为什么要选择不同形状的anchor box呢？直观印象是这样，我们将物体与anchor box进行比较，看看更像哪个anchor box的形状，和anchor box更像的物体倾向于被识别为anchor box代表的物体形状。例如anchor box1 更像行人的形状，而anchor box2 更像汽车的形状。
   那么如何定义、如何判断物体具体相似的形状呢？

3. IOU(交并比)
   定义是两个box的交集面积和并集面积的比值。就是预测框和真实框相近的大小。
4. 分类阈值过滤
   现在我们要为阈值进行过滤，我们要去掉一些预测值低于预设值的锚框。
5. 非最大值抑制
   即使是我们通过阈值来过滤了一些得分较低的分类，但是我们依旧会有很多的锚框被留了下来，第二个过滤器就是让下图左边变为右边，我们叫它非最大值抑制
   现在我们要实现非最大值抑制函数，关键步骤如下：
   1. 选择分值高的锚框
   2. 计算与其他框的重叠部分，并删除与iou_threshold相比重叠的框。
   3. 返回第一步，直到不再有比当前选中的框得分更低的框。
      TensorFlow有两个内置函数用于实现非最大抑制(tf.image.non_max_suppression和K.gather)

假设我们已经训练出了YOLO的模型，
首先输入待检测的图片，对图片进行一系列的处理，使得图片的规格符合数据集的要求。
第二，通过模型计算获得预测输出，假如使用的是19*19的grid cell数目，5个anchor box， 80个分类，于是输出的纬度为（1，19，19，5，80+5），
第三，对于输出值进行处理，过滤掉得分低的值，输出值中的Pc 在原论文中被称为confidence 而C被称为probs，得分为confidence * probs，可以看出，所谓的得分就是含有目标的概率值。
第四，同一个物体可能会有多个grid cell预测到，那么同一个物体就会有多个bouding box，我们需要留下具有最高pc值的预测值，将其他的预测值过滤掉。如何判断多个bounding box预测的是同一个物体呢，这里就需要使用IOU算法。最后得到的值就是图片中被预测的目标的类型和位置值，再经过一系列计算和转换变成图片上的真实坐标值，使用工具画到原图上。第四步中的筛选过程被称为Non-max suppression算法。
这就是YOLO算法中的预测部分，但至于训练部分，还有待学习，现在只能说了解甚浅。

8.21(人脸识别 and 风格转化)

人脸识别

1. 三元组损失函数 我们将使用三元组图像 （A， P， N） （A，P，N） （A，P，N）进行训练，A是“Anchor”，是一个人的图像，P是“Positive”，是相对于“Anchor”的同一个人的另外一张图像，N是“Negative”，是相对于“Anchor”的不同的人的另外一张图像。

$$\mathcal{J} = \sum_{i=1}^{m} \left[|| f(A^{(i)}) - f(P^{(i)}) ||_2^2 - || f(A^{(i)}) - f(N^{(i)}) ||_2^2 + \alpha \right]_+$$

   代码如下：

```
def triplet_loss(y_true, y_pred, alpha = 0.2):
 """
 根据公式（4）实现三元组损失函数

 参数：
     y_true -- true标签，当你在Keras里定义了一个损失函数的时候需要它，但是这里不需要。
     y_pred -- 列表类型，包含了如下参数：
         anchor -- 给定的“anchor”图像的编码，维度为(None,128)
         positive -- “positive”图像的编码，维度为(None,128)
         negative -- “negative”图像的编码，维度为(None,128)
     alpha -- 超参数，阈值
```

```
    返回：
        loss -- 实数，损失的值
    """
    #获取anchor, positive, negative的图像编码
    anchor, positive, negative = y_pred[0], y_pred[1], y_pred[2]

    #第一步：计算"anchor" 与 "positive"之间编码的距离，这里需要使用axis=-1
    pos_dist =
tf.reduce_sum(tf.square(tf.subtract(anchor,positive)),axis=-1)

    #第二步：计算"anchor" 与 "negative"之间编码的距离，这里需要使用axis=-1
    neg_dist =
tf.reduce_sum(tf.square(tf.subtract(anchor,negative)),axis=-1)

    #第三步：减去之前的两个距离，然后加上alpha
    basic_loss = tf.add(tf.subtract(pos_dist,neg_dist),alpha)

    #通过取带零的最大值和对训练样本的求和来计算整个公式
    loss = tf.reduce_sum(tf.maximum(basic_loss,0))

    return loss
```

2. 编码为128位的向量 我们的输出为128位的向量，而并不是softmax处理后的向量。
   而我们最后则是把数据库的人物输出到字典并记录，到时在系统上只重新把采集 到的人脸输入，把输出和数据库里的对比，就可以得出是否是正确的人。

风格转化

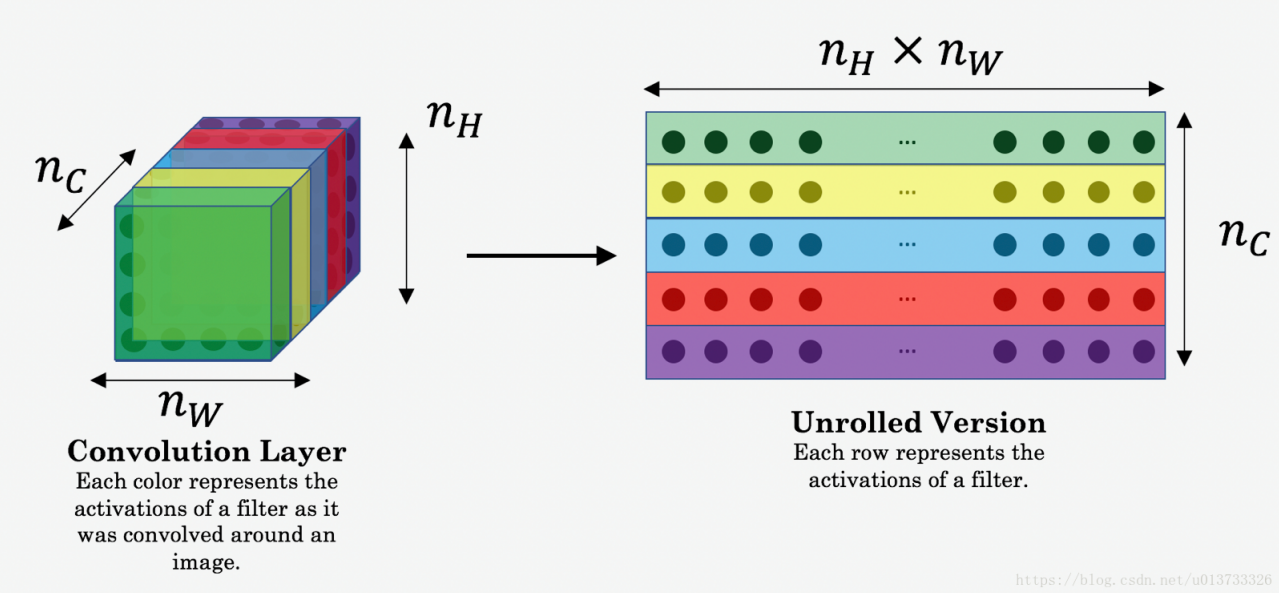
我们可以看看下面的图片

这就是神经风格转化

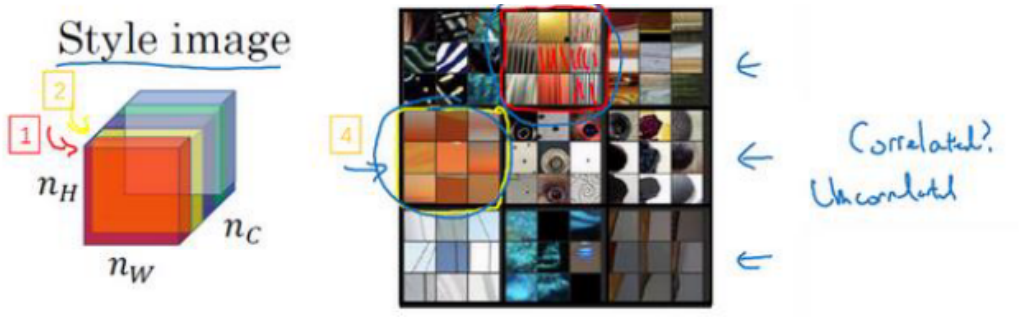
1. 计算内容损失　正如我们在视频中看到的，浅层的一个卷积网络往往检测到较低层次的特征，如边缘和简单的纹理，更深层往往检测更高层次的特征，如更复杂的纹理以及对象分类等。我们希望“生成的”图像G具有与输入图像C相似的内容。假设我们选择了一些层的激活来表示图像的内容，在实践中，如果你在网络中间选择一个层——既不太浅也不太深，你会得到最好的的视觉结果。（当你完成了这个练习后，你可以用不同的图层进行实验，看看结果是如何变化的。）

假设你选择了一个特殊的隐藏层，现在，将图像C作为已经训练好的VGG网络的输入，然后进行前向传播。让$a^{(C)}$成为你选择的层中的隐藏层激活（在视频中吴恩达老师写作$a^{[l](C)}$,但在这里我们将去掉上标[l]以简化符号），激活值为$n_H \times n_W \times n_C$的张量。然后用图像G重复这个过程：将G设置为输入数据，并进行前向传播，让$a^{(G)}$成为相应的隐层激活，我们将把内容成本函数定义为：

$$J_{content}(C,G) = \frac{1}{4 \times n_H \times n_W \times n_C} \sum_{所有条目} (a^{(C)} - a^{(G)})^2 \tag{1}$$



2. 计算风格损失



我们来看一个例子，这是之前视频中的一个可视化例子，它来自一篇论文，作者是 **Matthew Zeile** 和 **Rob Fergus** 我之前有提到过。我们知道，这个红色的通道（编号 1）对应的是这个神经元，它能找出图片中的特定位置是否含有这些垂直的纹理（编号 3），而第二个通道也就是黄色的通道（编号 2），对应这个神经元（编号 4），它可以粗略地找出橙色的区域。什么时候两个通道拥有高度相关性呢？如果它们有高度相关性，那么这幅图片中出现垂直纹理的地方（编号 2），那么这块地方（编号 4）很大概率是橙色的。如果说它们是不相关的，又是什么意思呢？显然，这意味着图片中有垂直纹理的地方很大概率不是橙色的。而相关系数描述的就是当图片某处出现这种垂直纹理时，该处又同时是橙色的可能性。

相关系数这个概念为你提供了一种去测量这些不同的特征的方法，比如这些垂直纹理，这些橙色或是其他的特征去测量它们在图片中的各个位置同时出现或不同时出现的频率。

$$G^{[l](S)}_{kk'} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a^{[l](S)}_{i,j,k} a^{[l](S)}_{i,j,k'}$$

代码如下：

```
def gram_matrix(A):
    """
    Argument:
    A -- matrix of shape (n_C, n_H*n_W)

    Returns:
    GA -- Gram matrix of A, of shape (n_C, n_C)
    """

    ### START CODE HERE ### (≈1 line)
    GA = tf.matmul(A, tf.transpose(A))
    ### END CODE HERE ###

    return GA
```

```
def compute_style_cost(model, STYLE_LAYERS):
 """
 Computes the overall style cost from several chosen layers

 Arguments:
 model -- our tensorflow model
 STYLE_LAYERS -- A python list containing:
                     - the names of the layers we would like to
extract style from
                     - a coefficient for each of them

 Returns:
 J_style -- tensor representing a scalar value, style cost defined
above by equation (2)
 """

 # initialize the overall style cost
 J_style = 0

 for layer_name, coeff in STYLE_LAYERS:

     # Select the output tensor of the currently selected layer
     out = model[layer_name]

     # Set a_S to be the hidden layer activation from the layer we
have selected, by running the session on out
     a_S = sess.run(out)

     # Set a_G to be the hidden layer activation from same layer.
Here, a_G references model[layer_name]
     # and isn't evaluated yet. Later in the code, we'll assign the
image G as the model input, so that
     # when we run the session, this will be the activations drawn
from the appropriate layer, with G as input.
```

```
        a_G = out

        # Compute style_cost for the current layer
        J_style_layer = compute_layer_style_cost(a_S, a_G)

        # Add coeff * J_style_layer of this layer to overall style cost
        J_style += coeff * J_style_layer

    return J_style




    def compute_layer_style_cost(a_S, a_G):
        """
        Arguments:
        a_S -- tensor of dimension (1, n_H, n_W, n_C), hidden layer
activations representing style of the image S
        a_G -- tensor of dimension (1, n_H, n_W, n_C), hidden layer
activations representing style of the image G

        Returns:
        J_style_layer -- tensor representing a scalar value, style cost
defined above by equation (2)
        """

        ### START CODE HERE ###
        # Retrieve dimensions from a_G (≈1 line)
        m, n_H, n_W, n_C = a_G.get_shape().as_list()

        # Reshape the images to have them of shape (n_H*n_W, n_C) (≈2
lines)
        a_S = tf.transpose(tf.reshape(a_S,(n_H*n_W,n_C)))
        a_G = tf.transpose(tf.reshape(a_G,(n_H*n_W,n_C)))

        # Computing gram_matrices for both images S and G (≈2 lines)
        GS = gram_matrix(a_S)
        GG = gram_matrix(a_G)

        # Computing the loss (≈1 line)
        J_style_layer =
(1/(4*n_H**2*n_W**2*n_C**2))*tf.reduce_sum(tf.square(tf.subtract(GS,GG
)))

        ### END CODE HERE ###

        return J_style_layer
```