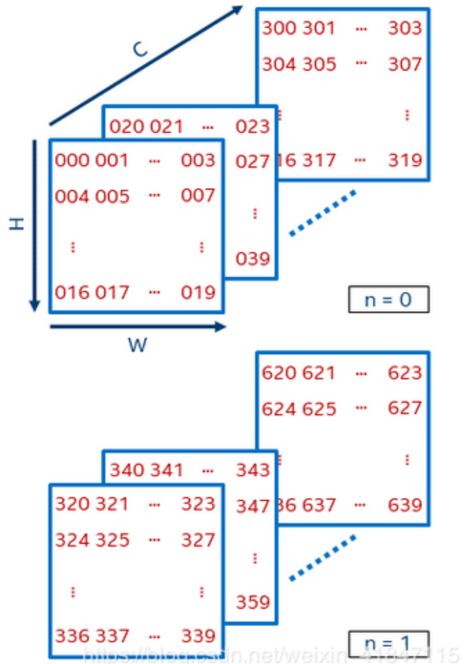
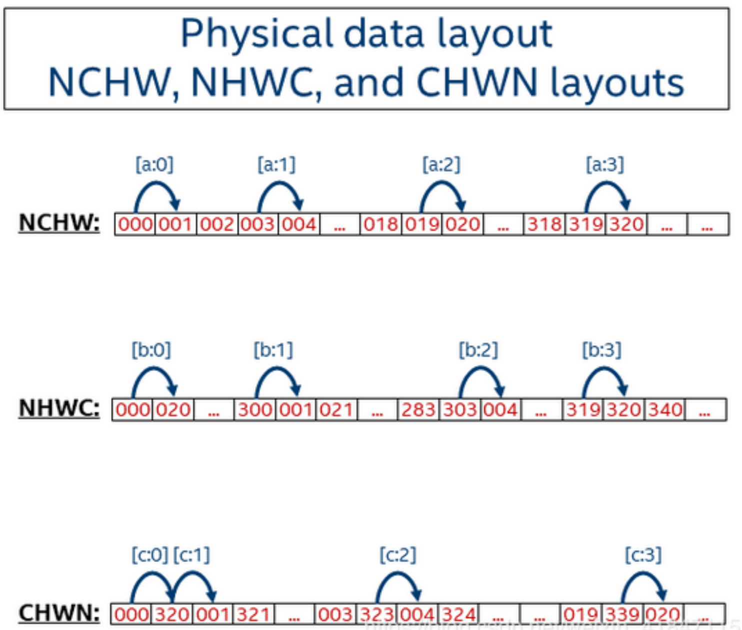
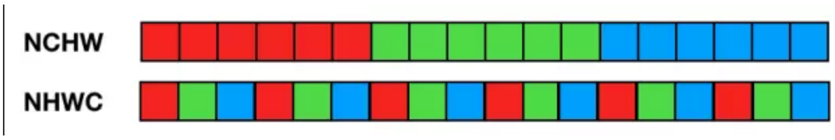
N代表数量， C代表channel，H代表高度，W代表宽度.



NCHW其实代表的是[W H C N]，第一个元素是000，第二个元素是沿着w方向的，即001，这样下去002 003，再接着呢就是沿着H方向，即004 005 006 007...这样到09后，沿C方向，轮到了020，之后021 022 ...一直到319，然后再沿N方向。
NHWC的话以此类推，代表的是[C W H N]，第一个元素是000，第二个沿C方向，即020，040, 060..一直到300，之后沿W方向，001 021 041 061...301..到了303后，沿H方向，即004 024 .。。304.。最后到了319，变成N方向，320,340....



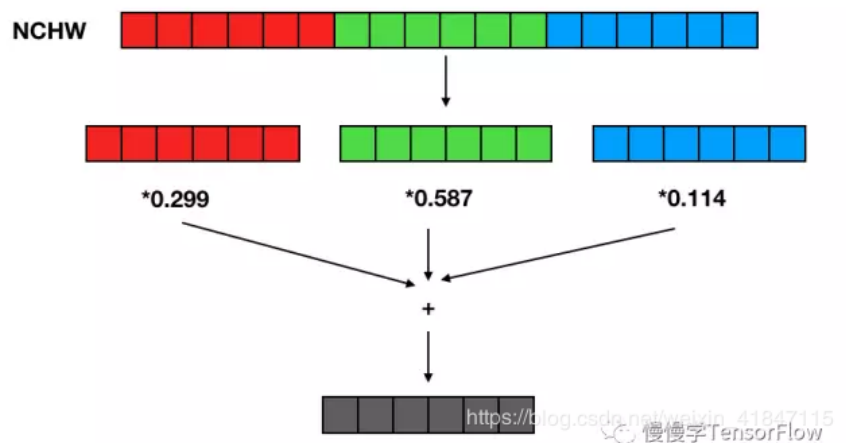
data_format 默认值为 "NHWC。其中 N 表示这批图像有几张，H 表示图像在竖直方向有多少像素，W 表示水平方向像素数，C 表示通道数（例如黑白图像的通道数 C = 1，而 RGB 彩色图像的通道数 C = 3）。为了便于演示，我们后面作图均使用 RGB 三通道图像。



NCHW 中，C 排列在外层，每个通道内像素紧挨在一起，即 'RRRRRRGGGGGGBBBBBB' 这种形式。

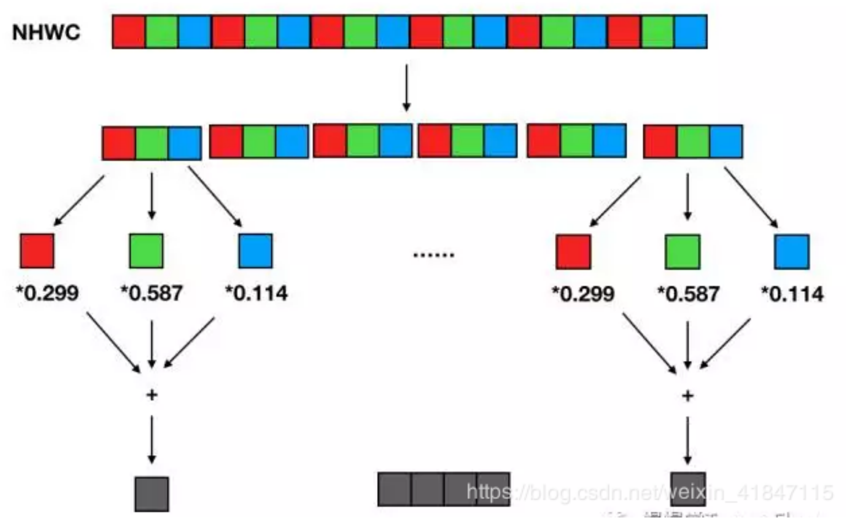
NHWC 格式，C 排列在最内层，多个通道对应空间位置的像素紧挨在一起，即 'RGBRGBRGBRGBRGBRGB' 这种形式。

如果我们需要对图像做彩色转灰度计算，NCHW 计算过程如下：



即 R 通道所有像素值乘以 0.299，G 通道所有像素值乘以 0.587，B 通道所有像素值乘以 0.114，最后将三个通道结果相加得到灰度值。

相应地，NHWC 数据格式的彩色转灰度计算过程如下：



输入数据分成多个(R, G, B) 像素组，每个像素组中 R 通道像素值乘以 0.299，G 通道像素值乘以 0.587，B 通道像素值乘以 0.114 后相加得到一个灰度输出像素。将多组结果拼接起来得到所有灰度输出像素。

以上使用两种数据格式进行 RGB -> 灰度计算的复杂度是相同的，**区别在于访存特性。**

通过两张图对比可以发现，NHWC 的访存局部性更好（每三个输入像素即可得到一个输出像素），NCHW 则必须等所有通道输入准备好才能得到最终输出结果，需要占用较大的临时空间。

在 CNN 中常常见到 1x1 卷积（例如：用于移动和嵌入式视觉应用的 MobileNets），也是每个输入 channel 乘一个权值，然后将所有 channel 结果累加得到一个输出 channel。如果使用 NHWC 数据格式，可以将卷积计算简化为矩阵乘计算，即 1x1 卷积核实现了每个输入像素组到每个输出像素组的线性变换。

TensorFlow 为什么选择 NHWC 格式作为默认格式？因为早期开发都是基于 CPU，使用 NHWC 比 NCHW 稍快一些（不难理解，NHWC 局部性更好，cache（缓存） 利用率高）。

NCHW 则是 Nvidia cuDNN 默认格式，使用 GPU 加速时用 NCHW 格式速度会更快（也有个别情况例外）。

最佳实践：

设计网络时充分考虑两种格式，最好能灵活切换，在 GPU 上训练时使用 NCHW 格式，在 CPU 上做预测时使用 NHWC 格式。

在不同的硬件加速的情况下，选用的类型不同，在intel GPU加速的情况下，因为GPU对于图像的处理比较多，希望在访问同一个channel的像素是连续的，一般存储选用NCHW，这样在做CNN的时候，在访问内存的时候就是连续的了，比较方便。

https://github.com/apache/incubator-tvm/blob/3e3ccce1135c25dd1d99dc7c2b8ff589c93ee7ea/topi/python/topi/nn/conv2d.py

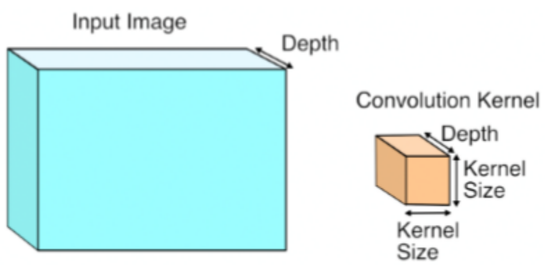
```
def conv2d(input, filter, strides, padding, dilation, layout='NCHW', out_dtype=None):
 """
 output : tvm.te.Tensor
 4-D with shape [batch, out_channel, out_height, out_width]
 """
 # search platform specific declaration first
 # default declaration
 if layout == 'NCHW':
 return conv2d_nchw(input, filter, strides, padding, dilation, out_dtype)
  if layout == 'HWCN':
  return conv2d_hwcn(input, filter, strides, padding, dilation, out_dtype)
  if layout == 'NHWC':
  return conv2d_nhwc(input, filter, strides, padding, dilation, out_dtype)
  raise ValueError("not support this layout {} yet".format(layout))
```

```
def conv2d_nchw(Input, Filter, stride, padding, dilation, out_dtype=None):
 """Convolution operator in NCHW layout.
 Output : tvm.te.Tensor
 4-D with shape [batch, out_channel, out_height, out_width]
 """
 if out_dtype is None:
 out_dtype = Input.dtype
 assert isinstance(stride, int) or len(stride) == 2
 assert isinstance(dilation, int) or len(dilation) == 2
  if isinstance(stride, int):
  stride_h = stride_w = stride
  else:
  stride_h, stride_w = stride

  if isinstance(dilation, int):
  dilation_h = dilation_w = dilation
```

```
else:
dilation_h, dilation_w = dilation

batch, in_channel, in_height, in_width = Input.shape
num_filter, channel, kernel_h, kernel_w = Filter.shape
# compute the output shape
dilated_kernel_h = (kernel_h - 1) * dilation_h + 1
dilated_kernel_w = (kernel_w - 1) * dilation_w + 1
pad_top, pad_left, pad_down, pad_right = get_pad_tuple(padding, (dilated_kernel_h, dilated_kernel_w))
out_channel = num_filter
out_height = simplify((in_height - dilated_kernel_h + pad_top + pad_down) // stride_h + 1)
out_width = simplify((in_width - dilated_kernel_w + pad_left + pad_right) // stride_w + 1)
# compute graph
pad_before = [0, 0, pad_top, pad_left]
pad_after = [0, 0, pad_down, pad_right]
temp = pad(Input, pad_before, pad_after, name="pad_temp")
rc = te.reduce_axis((0, in_channel), name='rc')
ry = te.reduce_axis((0, kernel_h), name='ry')
rx = te.reduce_axis((0, kernel_w), name='rx')
return te.compute((batch, out_channel, out_height, out_width),
lambda nn, ff, yy, xx:
te.sum(temp[nn, rc, yy * stride_h + ry * dilation_h,xx * stride_w + rx * dilation_w].astype(out_dtype)
*Filter[ff, rc, ry, rx].astype(out_dtype),axis=[rc, ry, rx]),
tag="conv2d_nchw")
```



https://tvm.apache.org/docs/api/python/relay/nn.html?highlight=conv2d#tvm.relay.nn.conv2d

`tvm.relay.nn.conv2d`*(data, weight, strides=(1, 1), padding=(0, 0), dilation=(1, 1), groups=1, channels=None, kernel_size=None, data_layout='NCHW', kernel_layout='OIHW', out_layout='', out_dtype='')*

2D convolution.

This operator takes the weight as the convolution kernel and convolves it with data to produce an output.

In the default case, where the data_layout is *NCHW* and kernel_layout is *OIHW* , **conv2d** takes in

a data Tensor with shape *(batch_size, in_channels, height, width)*, and

a weight Tensor with shape *(channels, in_channels, kernel_size[0], kernel_size[1])*

to produce an output Tensor with the following rule:

$$out[b,c,y,x]=\sum_{dy,dx,k} data[b,k,strides[0]*y+dy,strides[1]*x+dx]*weight[c,k,dy,dx]$$

$$\mathrm{out}[b,c,y,x] = \sum_{dy,dx,k} \mathrm{data}[b,k,\mathrm{strides}[0]*y+dy,\mathrm{strides}[1]*x+dx]*\mathrm{weight}[c,k,dy,dx]$$

Padding and dilation are applied to data and weight respectively before the computation. This operator accepts data layout specification. Semantically, the operator will convert the layout to the canonical layout (*NCHW* for data and *OIHW* for weight), perform the computation, then convert to the out_layout.

Parameters

- **data** (*tvm.relay.Expr*) – The input data to the operator.
- **weight** (*tvm.relay.Expr*) – The weight expressions.
- **strides** (*Optional[int, Tuple[int]]*) – The strides of convolution.

- **padding** (*Optional[int, Tuple[int]]*) – The padding of convolution on both sides of inputs before convolution.
- **dilation** (*Optional[int, Tuple[int]]*) – Specifies the dilation rate to be used for dilated convolution.
- **groups** (*Optional[int]*) – Number of groups for grouped convolution.
- **channels** (*Optional[int]*) – Number of output channels of this convolution.
- **kernel_size** (*Optional[int, Tuple[int]]*) – The spatial of the convolution kernel.
- **data_layout** (*Optional[str]*) – Layout of the input.
- **kernel_layout** (*Optional[str]*) – Layout of the weight.
- **out_layout** (*Optional[str]*) – Layout of the output, by default, out_layout is the same as data_layout
- **out_dtype** (*Optional[str]*) – Specifies the output data type for mixed precision **conv2d**.

Returns

result – The computed result.

Return type

tvm.relay.Expr

```
@tf_export('nn.conv2d')
def conv2d(input, filter, strides, padding, use_cudnn_on_gpu=True, data_format="NHWC", dilations=[1, 1, 1, 1], name=None):
r"""Computes a 2-D convolution given 4-D `input` and `filter` tensors.
Given an input tensor of shape `[batch, in_height, in_width, in_channels]`
and a filter / kernel tensor of shape
`[filter_height, filter_width, in_channels, out_channels]`, this op
performs the following:
1. Flattens the filter to a 2-D matrix with shape
 `[filter_height * filter_width * in_channels, output_channels]`.
 2. Extracts image patches from the input tensor to form a *virtual*
  tensor of shape `[batch, out_height, out_width,
  filter_height * filter_width * in_channels]`.
 3. For each patch, right-multiplies the filter matrix and the image patch
  vector.
 In detail, with the default NHWC format,
  output[b, i, j, k] =
  sum_{di, dj, q} input[b, strides[1] * i + di, strides[2] * j + dj, q] *
  filter[di, dj, q, k]
 Must have `strides[0] = strides[3] = 1`. For the most common case of the same
 horizontal and vertices strides, `strides = [1, stride, stride, 1]`.
 Args:
  input: A `Tensor`. Must be one of the following types: `half`, `bfloat16`, `float32`, `float64`.
  A 4-D tensor. The dimension order is interpreted according to the value
  of `data_format`, see below for details.
  filter: A `Tensor`. Must have the same type as `input`.
  A 4-D tensor of shape
  `[filter_height, filter_width, in_channels, out_channels]`
  strides: A list of `ints`.
  1-D tensor of length 4. The stride of the sliding window for each
  dimension of `input`. The dimension order is determined by the value of
  `data_format`, see below for details.
  padding: A `string` from: `"SAME", "VALID"`.
  The type of padding algorithm to use.
  use_cudnn_on_gpu: An optional `bool`. Defaults to `True`.
  data_format: An optional `string` from: `"NHWC", "NCHW"`. Defaults to `"NHWC"`.
  Specify the data format of the input and output data. With the
  default format "NHWC", the data is stored in the order of:
  [batch, height, width, channels].
  Alternatively, the format could be "NCHW", the data storage order of:
  [batch, channels, height, width].
```

```
    dilations: An optional list of `ints`. Defaults to `[1, 1, 1, 1]`.
    1-D tensor of length 4. The dilation factor for each dimension of
    `input`. If set to k > 1, there will be k-1 skipped cells between each
    filter element on that dimension. The dimension order is determined by the
    value of `data_format`, see above for details. Dilations in the batch and
    depth dimensions must be 1.
    name: A name for the operation (optional).
  Returns:
    A `Tensor`. Has the same type as `input`.
  """
```