卷积层的输出：

Summary. To summarize, the Conv Layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P.
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry)
  - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases.

如果 W2 或者 H2 计算得到不为整数：
not an integer, indicating that the neurons don't "fit" neatly and symmetrically across the input. Therefore, this setting of the hyperparameters is considered to be invalid, and a ConvNet library could throw an exception or zero pad the rest to make it fit, or crop the input to make it fit, or something. >>

使用 *tf.nn.conv2d* 或 *tf.layers.conv2d* 作为卷积函数，当不为整时根据 padding 取值（默认为"valid"）由下计算：

`padding`： `"valid"` 或者 `"same"`（不区分大小写）。`"valid"` 表示不够卷积核大小的块就丢弃，`"same"` 表示不够卷积核大小的块就补0。

`"valid"` 的输出形状为

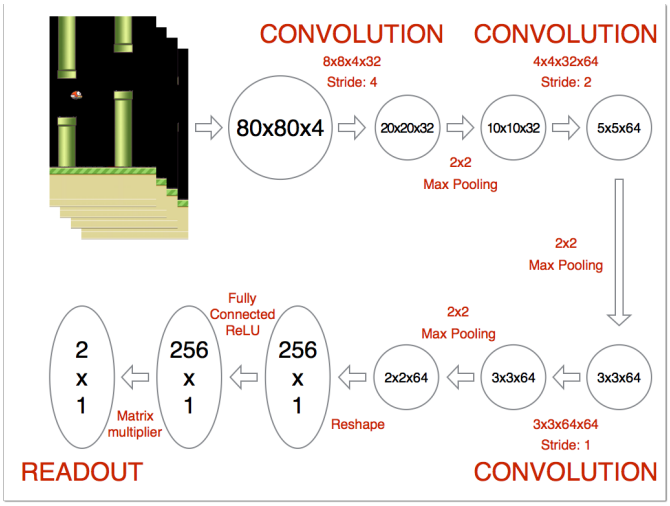
$$L_{new} = ceil(\frac{L - F + 1}{S})$$

`"same"` 的输出形状为

$$L_{new} = ceil(\frac{L}{S})$$

其中，L 为输入的 size（高或宽），F 为 filter 的 size，S 为 strides 的大小，$ceil()$ 为向上取整。

上述公式**池化**也适用。
比如 DQN 解决 `Flappy bird` 中的 DQN 网络，其中卷积和池化用到的函数 `padding="same"`，可依次得到每后一层的 size。 >>



再比较下 tf.nn.conv2d 或 tf.layers.conv2d 两个函数，拿上图第一层卷积为例，先定义输入
```
input = tf.placeholder(tf.float32, shape=[None, 80, 80, 4])
```
若使用 tf.nn.conv2d：
```
kernel = tf.truncated_normal([8, 8, 4, 32], stddev = 0.01)
h1 = tf.nn.conv2d(input, kernel, strides = [1, 4, 4, 1], padding = "SAME")
h_conv1 = tf.nn.relu(h1)
```
若使用 tf.layers.conv2d：
```
h_conv1 = tf.layers.conv2d(input, 32, 8, strides=4, padding = "SAME", activation=tf.nn.relu)
```
或补全：
```
h_conv1 = tf.layers.conv2d(input, 32, (8, 8), strides=(4, 4), padding = "SAME", activation=tf.nn.relu)
```

反卷积的输出：
反卷积的过程和卷积操作相反，因此由卷积层的输入输出关系可以得到反卷积层的输入输出关系：

If you assume the following notation, `output = o`, `input = i`, `kernel = k`, `stride = s`, `padding = p`, the shape of the output will be:

$$o = s(i - 1) + k - 2p.$$

但是也存在如上正算卷积所得非整的情况，因此可以直接根据之前 padding 的不同取值对应的公式来求解输出 size(此时 Lnew 为输入，L 为输出)
举一例，先定义输入：

```
a = np.array([[1,1],[2,2]], dtype=np.float32)
a = np.reshape(a, [1,2,2,1])    # input 须为 a 4-D tensor: [batch, in_height, in_width, in_channel]
x = tf.constant(a, dtype=tf.float32)
```

若使用 **tf.nn.conv2d_transpose** 函数，可以指定输出 size

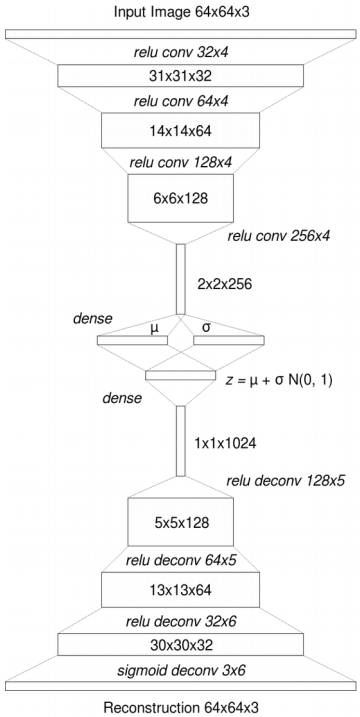
```
kernel = tf.constant(1.0, shape=[3,3,1,1])
upsample_x = tf.nn.conv2d_transpose(x, kernel, output_shape=(1,4,4,1), strides=(1,2,2,1), padding='SAME')
with tf.Session() as sess:
    tf.global_variables_initializer().run()
    print(sess.run(upsample_x))
```

或指定 *output_shape=(1,3,3,1)*，但一定要科学，在此例中输出的 size 只能为 3 或 4（padding="SAME"，反卷积的输出或卷积的输入为 2，因此反卷积的输出或卷积的输入只能为 3 或 4），如果令 output_shape=(1,5,5,1)则就会报错
若使用 **tf.layers.conv2d_transpose**(inputs,filters,kernel_size,strides=(1, 1))，不能指定输出 size，应该是按照最一般情况运行的，这里和指定(1,4,4,1)的结果相同

```
upsample_x = tf.layers.conv2d_transpose(x, 1, 3, strides=2, padding='SAME',
kernel_initializer=tf.ones_initializer())
```

通过上边的例子可以看到 tf.layer 相比与 tf.nn 库的函数更为简洁方便，一个可以集成一个层的全部定义，而 tf.nn 无法初始化 kernel，也无法在函数里直接给定激活函数。

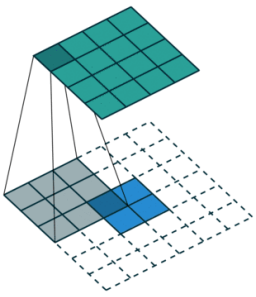
看下边为 VAE 模型，每个卷积层和反卷积层的 padding="VALID"，strides=2，tf.layers.conv2d 实现，可以分析其设计的合理性 >>

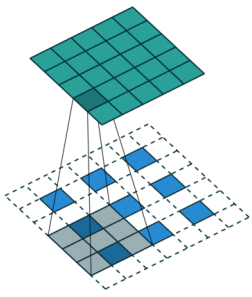


而 Keras 库函数 **keras.layers.Conv2DTranspose** 参数除了 padding 外还可以通过 output_padding 来填充输出，默认为 0，因此设计时需要注意。输出公式为

```
new_rows = ((rows - 1) * strides[0] + kernel_size[0] - 2 * padding[0] + output_padding[0])
new_cols = ((cols - 1) * strides[1] + kernel_size[1] - 2 * padding[1] + output_padding[1])
```

再说说 strides，当其值为 1 时外围填充，如下左图所示；当其大于 1 时为间隙填充，如下右图所示：





添加 Batch Normalization
Batch Normalization:

> First introduced in the paper: Accelerating Deep Network Training by Reducing Internal Covariate Shift.
>
> As the data flows through a deep network, the weights and parameters adjust those values, sometimes making the data too big or too small again - a problem the authors refer to as "internal covariate shift". By normalizing the data in each mini-batch, this problem is largely avoided. Batch Normalization normalizes each batch by both mean and variance reference.

通常放在神经网络的线性层与非线性层之间，来归一化激活函数的输入。来看看使用 keras 时分别在 Dense 与 Conv 层添加 BN >>

Dense layer

A normal **Dense** fully connected layer looks like this

```
model.add(layers.Dense(64, activation='relu'))
```

To make it Batch normalization enabled, we have to tell the Dense layer not using bias since it is not needed, it can save some calculation. Also, put the Activation layer after the `BatchNormalization()` layer

```
model.add(layers.Dense(64, use_bias=False))
model.add(layers.BatchNormalization())
model.add(Activation("relu"))
```

Conv2D layer

A normal Keras **Conv2D** layer can be defined as

```
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

Turing it to Batch normalized Conv2D layer, we add the `BatchNormalization()` layer similar to Dense layer above

```
model.add(layers.Conv2D(64, (3, 3), use_bias=False))
model.add(layers.BatchNormalization())
model.add(layers.Activation("relu"))
```

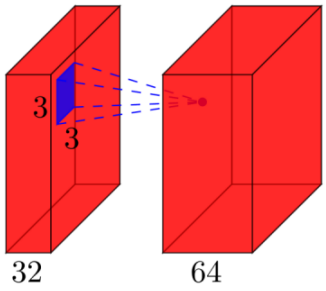
上例中 BatchNormalization 函数用的默认参数，具体地：
keras.layers.BatchNormalization(axis=-1, momentum=0.99, epsilon=0.001, ...) >>
axis:归一化的维度，对于 Conv2D 层若 data_format="channels_last", axis=-1
data_format="channels_first", axis=1
momentum: 移动均值和移动方差的动量
epsilon: 增加到方差的小的浮点数，以避免除以零

tensorflow 实现：
```
layer = tf.layers.dense(prev_layer, num_units, kernel_initializer=tf.contrib.layers.xavier_initializer(), activation=None)
layer = tf.layers.batch_normalization(layer, training=restore)  # training 参数表示该网络当前是否正在训练，告知 Batch Normalization 层是否应该更新或者使用均值或方差的分布信息，可根据 restore 值确定
layer = tf.nn.relu(layer)
```

添加 L2
Regularizers allow to apply penalties on layer parameters or layer activity during optimization. These penalties are incorporated in the loss function that the network optimizes. >>
L2 范数正则化在模型原损失函数基础上添加 L2 范数惩罚项，从而得到训练所需要最小化的函数。L2 范数正则化又叫权重衰减。权重衰减通过惩罚绝对值较大的模型参数为需要学习的模型增加了限制，这可能对过拟合有效。实际场景中，我们有时也在惩罚项中添加偏差元素的平方和。
Keras 对于常用的神经网络层函数都有统一的 API 接口来添加 L1 或者 L2 范数，如
```
from keras.regularizers import l2
from keras.models import Sequential
from keras.layers import Conv2D, Dense
model = Sequential()
model.add(Conv2D(filters=32, kernel_size=3, strides=2, activation='relu', kernel_regularizer=l2(0.001)))
model.add(Dense(64, kernel_regularizer=regularizers.l2(0.001))
```

防止过拟合：使用复杂度适合的模型（模型太过简单容易欠拟合，模型太过复杂容易过拟合）、增大训练量、权重衰减（L2 范数正则化）、dropout

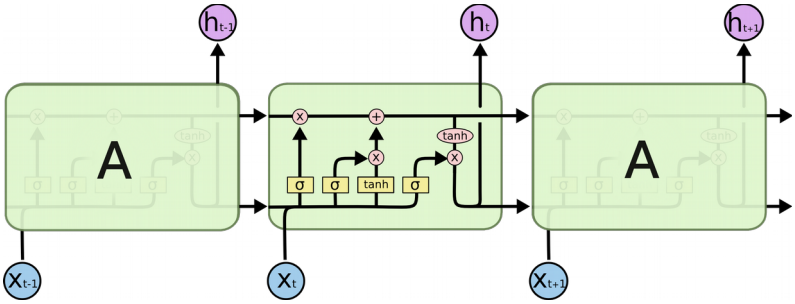
计算神经网络的参数个数
卷积层：(n*m*l+1)*k，其中滤波器尺寸为 n*m，输入深度为 l，输出深度为 k



如上图卷积层输入的深度(or feature map)是 32，输出的深度是 64，滤波器尺寸为为 3*3.
需要理解的是我们并没有简单地使用 3×3 的滤波器，而实际使用的是 3×3×32 的滤波器，因为我们的输入是 32 维。根据输出总共需要学习 64 种不同的 3×3×32 滤波器。因此权重的总数是 3×3×32×64，此外输出每层深度有一个偏差项，最后的参数总数为（3×3×32+1）×64。
池化层：0，执行如用最大值替换 2×2 领域的操作，因此无须在池化层学习任何参数
全连接层：(a+1)*b，其中 a 为输入结点数，b 为输出结点数
总的权重数是 a*b，此外每个输出结点都有个偏差，因此参数总数是(a+1)*b
来看这样一个识别 MNIST 手写数字的网络(输入为 28×28 的灰度图，channel first)：
输入*(1×28×28) → Conv|32|5×5|1 → Maxpool|2×2 → Conv|32|3×3|1 → Maxpool|2×2 → FC|256 → FC|10*
首先计算每层的输出，其中卷积层的 padding 采用默认值"VALID"，然后依次计算每层的参数求和得到网络的总参数：

```
  #  name                          size                  parameters
---  --------  ------------------------------    ------------------------
  0  input                       1x28x28                              0
  1  conv2d1   (28-(5-1))=24 -> 32x24x24    (5*5*1+1)*32   =     832
  2  maxpool1                    32x12x12                             0
  3  conv2d2   (12-(3-1))=10 -> 32x10x10    (3*3*32+1)*32  =   9'248
  4  maxpool2                     32x5x5                              0
  5  dense                           256    (32*5*5+1)*256 = 205'056
  6  output                           10    (256+1)*10     =   2'570
```

网络总的可学习参数为：832 + 9'248 + 205'056 + 2'570 = 217'706　　>>
LSTM 层：(a+b+1)*b*4，其中 a 为输入向量的维度，b 为输出向量的维度（等于 hidden units 的个数）



a+b:连接[h(t-1),x(t)]
+1: 偏差项，若无偏差项则不加
*4: 4 个神经网络（黄色框）{W_forget, W_input, W_output, W_cell}
最后一项 b 是存在偏差项的时候，在 keras.layers.LSTM 函数中是默认存在的。

```
from keras.models import Sequential
from keras.layers import LSTM
model = Sequential()
model.add(LSTM(units=256, input_dim=4096, input_length=16))
model.summary()
```

输出总的参数为 4457472
由公式计算：(4096+256+1)*256*4 = 4457472

损失函数
binary_crossentropy，对数损失函数，与 sigmoid（输出为单个结点）相对应的损失函数。
categorical_crossentropy，多分类的对数损失函数，与 softmax 分类器相对应的损失函数。

Keras 用法：
1. 限定张量 result 每个元素不大于 1：

```
from keras import backend as K
result = K.random_normal(shape=(2, 2), mean=0.0, stddev=1.0)
result = K.expand_dims(result, -1)  # shape=(2, 2, 1)
max_value = K.ones_like(result)
# 使用K.min(x, axis)比较：取张量中的最大值
result = K.concatenate([result, max_value], -1)  # shape=(2,2,2)
result = K.min(result, -1, keepdims=False)  # shape=(2,2)
# 使用K.minimum(x, y)比较：逐个元素比对两个张量的最小值
result = K.minimum(result, max_value)
# 打印结果
print(K.get_value(result))  # 或
print(K.eval(result))
```

2. 找出 result 中的最大元素：

```
max_element = K.get_value(K.max(K.flatten(result)))
```