

# Redes de Convolução

In [1]:

```
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

In [2]:

```
import tensorflow as tf
```

In [3]:

```
# reinicia grafo do tensorflow
def reset_graph(seed=42):
    tf.reset_default_graph()
    tf.set_random_seed(seed)
    np.random.seed(seed)
```

# Convoluçãoes

In [4]:

```
import skimage.measure # scikit-image
```

Para entendermos convolução e pooling, vamos inicialmente ver o que essas operações fazem em pequenos exemplos de imagens.

Para isso, vamos criar uma função para exibir matrizes como imagens em tons de cinza.

In [5]:

```
def plot_figs(lst):
    if len(lst) == 1:
        plt.matshow(lst[0], cmap = 'gray', interpolation='nearest')
    else:
        f, axes = plt.subplots(1, len(lst))
        for i, a in enumerate(axes):
            a.matshow(lst[i], cmap = 'gray', interpolation='nearest')
            a.set(aspect='equal')
```

E então exibir exemplos de imagens simples (ex0 e ex1) e de um kernel simples (k0):

In [6]:

```
ex0 = np.array([[0,0,0,1,0,0,0,0],
               [0,0,1,0,0,0,0,0],
               [0,1,0,0,0,0,0,0],
               [1,0,0,0,0,0,0,0],
               [0,0,0,0,1,0,0,0],
               [0,0,0,0,0,1,0,0],
               [0,0,0,0,0,0,1,0],
               [0,0,0,0,0,0,0,1]], dtype=np.float32)
```

In [7]:

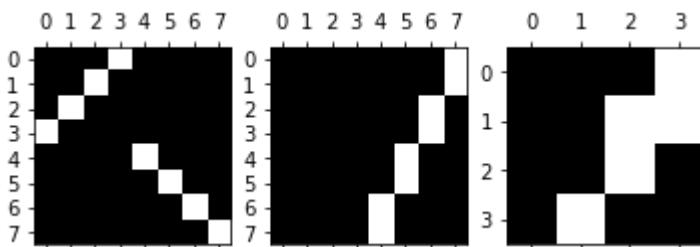
```
ex1 = np.array([[0,0,0,0,0,0,0,1],
               [0,0,0,0,0,0,0,1],
               [0,0,0,0,0,0,1,0],
               [0,0,0,0,0,0,1,0],
               [0,0,0,0,0,1,0,0],
               [0,0,0,0,0,1,0,0],
               [0,0,0,0,1,0,0,0],
               [0,0,0,0,1,0,0,0]], dtype=np.float32)
```

In [8]:

```
k0 = np.array([[0,0,0,1],
               [0,0,1,1],
               [0,0,1,0],
               [0,1,0,0]], dtype=np.float32)
```

In [9]:

```
plot_figs([ex0, ex1, k0])
```



O componente básico da operação de convolução é o produto (interno) entre a entrada ( $x$ ) e o kernel ( $w$ ), com saída opcionalmente transformada, por exemplo, para ter uma interpretação probabilística. Ou seja, ela pode ser representada como  $\sigma(x^T w)$ , onde  $\sigma$  pode ser a função sigmoid:

In [10]:

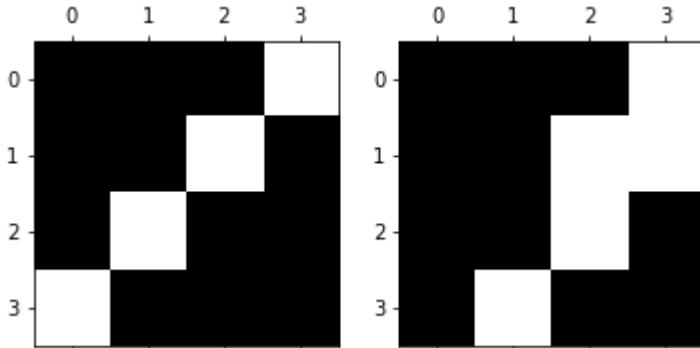
```
def conv0(ilist, k):
    sigmoid = lambda n: 1. / (1. + np.exp(-n))
    kr = k.reshape((-1, 1))
    s = 0
    for i in ilist:
        lstr = i.reshape((-1, 1))
        s += np.sum(np.multiply(lstr, kr))
    return sigmoid(s)
# return sigmoid(np.dot(lstr[i].T, kr))[0,0]
```

In [11]:

```
plot_figs([ex0[:4,:4], k0])
conv0([ex0[:4,:4]], k0)
```

Out[11]:

0.8807970779778823



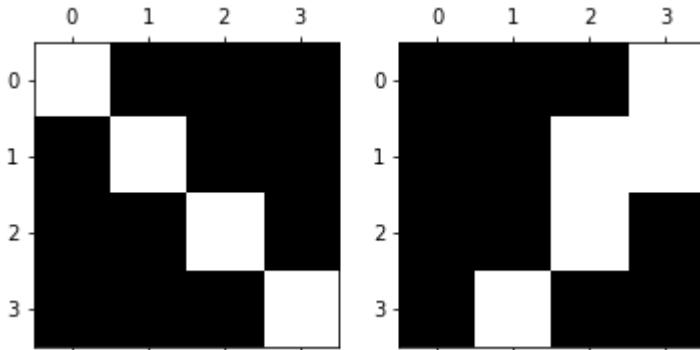
Ao aplicar conv0 para k0 e a parte superior esquerda de ex0, notamos que o padrão k0 é parcialmente observado em ex0 (probabilidade de 88%).

In [12]:

```
plot_figs([ex0[4:,4:], k0])
conv0([ex0[4:,4:]], k0)
```

Out[12]:

0.7310585786300049



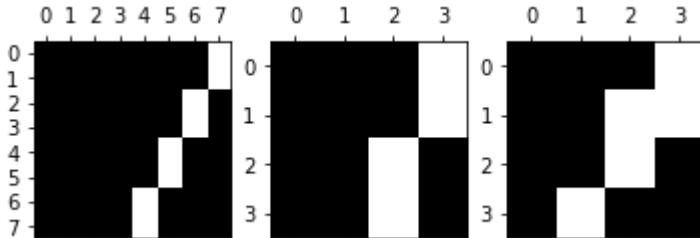
Ao aplicar conv0 para k0 e a parte inferior direira de ex0, notamos que o padrão k0 é observado com certeza menor (probabilidade de 73%), umas vez que houve coincidênciа em um único ponto.

In [14]:

```
ex1p = skimage.measure.block_reduce(ex1, (2,2), np.max)
plot_figs([ex1, ex1p, k0])
conv0([ex1p], k0)
```

Out[14]:

0.9525741268224334



Ao aplicar conv0 para k0 e uma versão *reduzida* de ex1, notamos que o padrão k0 é observado com certo grau de certeza (probabilidade de 95%).

Destes exemplos, podemos interpretar a operação de convolução como um problema de dizer se um padrão pode ou não ser visto em uma imagem. Podemos ver também que reduzir a imagem (uma operação de *pooling*) pode tornar um padrão visível.

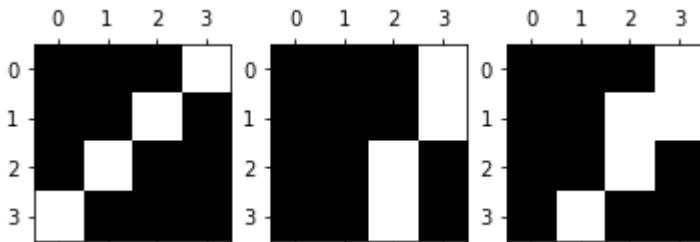
Agora vamos ver a aplicação de um kernel sobre mais de uma entrada:

In [15]:

```
plot_figs([ex0[:4,:4], ex1p, k0])
conv0([ex0[:4,:4], ex1p], k0)
```

Out[15]:

0.9933071490757153



A seguir, vamos ver a aplicação da operação de convolução completa de um kernel sobre uma imagem.

Para isso vamos usar o operador de convolução conv2D do tensorflow. Os parâmetros deste operador são a imagem de entrada (tensor 4D [número de batches, altura, largura, número de canais]), o conjunto de kernels a serem aplicados (tensor 4D [altura, largura, canais de entrada, canais de saída]), os deslocamentos a serem aplicados (no conjunto de batches, nos pixels da altura, pixels da largura e canais de entrada) e o tipo de *padding* a ser realizado ('VALID' = sem padding ou 'SAME' = tentar manter o mesmo padding em cada margem).

In [16]:

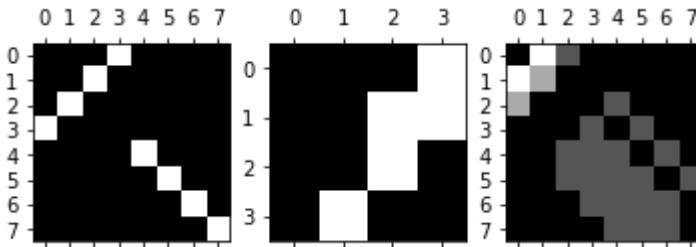
```
def tfconv(imgs, kernels):
    reset_graph()

    # batch, alt, larg, canais entrada
    X = tf.constant(imgs, dtype=tf.float32)
    feature_maps = tf.constant(kernels)
    convolution = tf.nn.conv2d(X, feature_maps, strides=[1,1,1,1],
                               padding="SAME")

    with tf.Session() as s:
        output = convolution.eval(feed_dict={X: imgs})
    return output
```

In [17]:

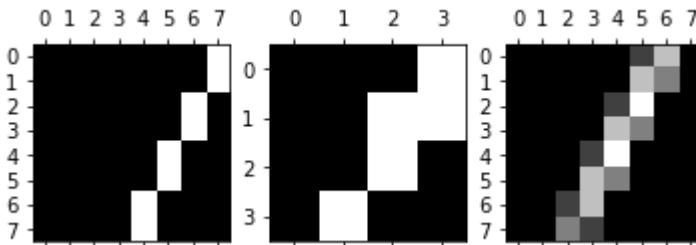
```
# alt, larg, canais de entrada, canais de saída
fmap = k0.reshape(k0.shape[0], k0.shape[1], 1, 1)
output = tfconv(ex0.reshape(1, ex0.shape[0], ex0.shape[1], 1), fmap)
plot_figs([ex0, k0, output[0, :, :, 0]])
```



Ao aplicar convolução do kernel k0 em ex0, notamos que o padrão k0 é observado na parte superior esquerda da imagem, mas não muito claramente na parte inferior esquerda. Esta é a forma como pode ser interpretada a imagem resultante da convolução.

In [18]:

```
output = tfconv(ex1.reshape(1, ex1.shape[0], ex1.shape[1], 1), fmap)
plot_figs([ex1, k0, output[0, :, :, 0]])
```



Ao aplicar convolução do kernel k0 em ex1, notamos que o padrão k0 parece ser observado ao longo de ex1, ao lado esquerdo (em particular, mais na parte superior que na parte inferior da imagem).

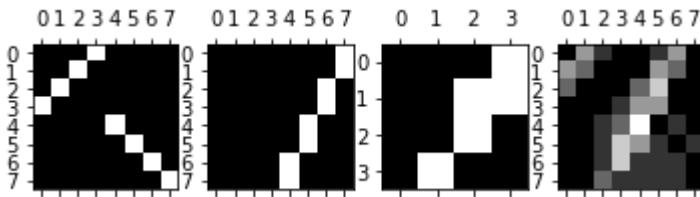
Na prática, contudo, vamos usar kernels não apenas como filtros, mas também como combinadores lineares. Para tanto, basta que a entrada tenha mais canais que a saída, como no exemplo:

In [19]:

```
# alt, larg, canais de entrada, canais de saída
fmap = np.zeros((k0.shape[0], k0.shape[1], 2, 1), dtype = np.float32)
fmap[:, :, 0, 0] = k0
fmap[:, :, 1, 0] = k0
# batch, altura, largura, canais de entrada
icanais = np.zeros((1, ex0.shape[0], ex0.shape[1], 2), dtype = np.float32)
icanais[0, :, :, 0] = ex0
icanais[0, :, :, 1] = ex1
```

In [20]:

```
output = tfconv(icanais, fmap)
plot_figs([ex0, ex1, k0, output[0, :, :, 0]])
```



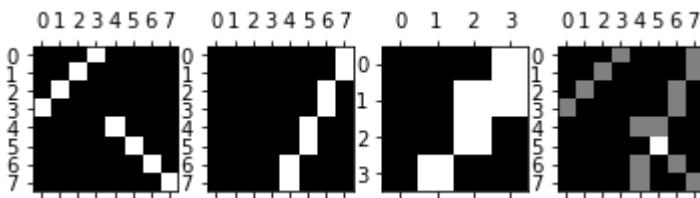
Outra observação interessante é que, embora um filtro  $1 \times 1$  não tenha efeito nenhum quando aplicado sobre uma única entrada, ele se comporta como um combinador linear quando aplicado a múltiplas:

In [21]:

```
# alt, larg, canais de entrada, canais de saída
fmap = np.zeros((1, 1, 2, 1), dtype = np.float32)
fmap[:, :, 0, 0] = np.array([1])
fmap[:, :, 1, 0] = np.array([1])
# batch, altura, largura, canais de entrada
icanais = np.zeros((1, ex0.shape[0], ex0.shape[1], 2), dtype = np.float32)
icanais[0, :, :, 0] = ex0
icanais[0, :, :, 1] = ex1
```

In [22]:

```
output = tfconv(icanais, fmap)
plot_figs([ex0, ex1, k0, output[0, :, :, 0]])
```



Finalmente, vamos ver o efeito de padrões (kernels) mais complexos em uma imagem real.

In [23]:

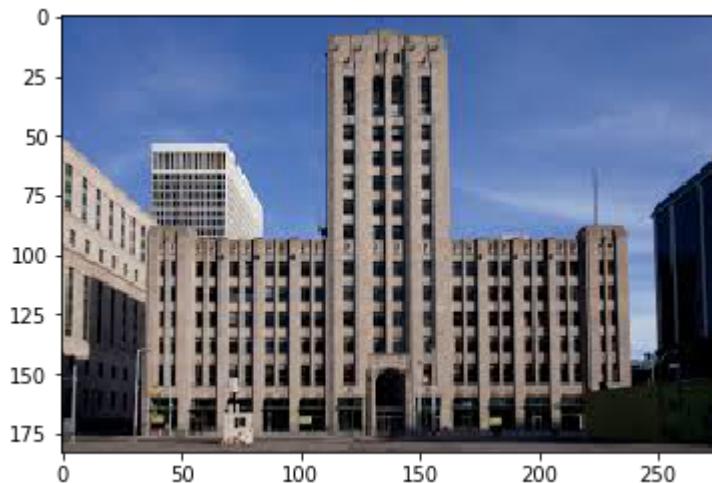
```
im = Image.open('images/building.jpeg')
image = np.asarray(im)
```

In [24]:

```
plt.imshow(image)
```

Out[24]:

```
<matplotlib.image.AxesImage at 0x7f7fc4041a90>
```



In [25]:

```
image.shape
```

Out[25]:

```
(183, 275, 3)
```

In [26]:

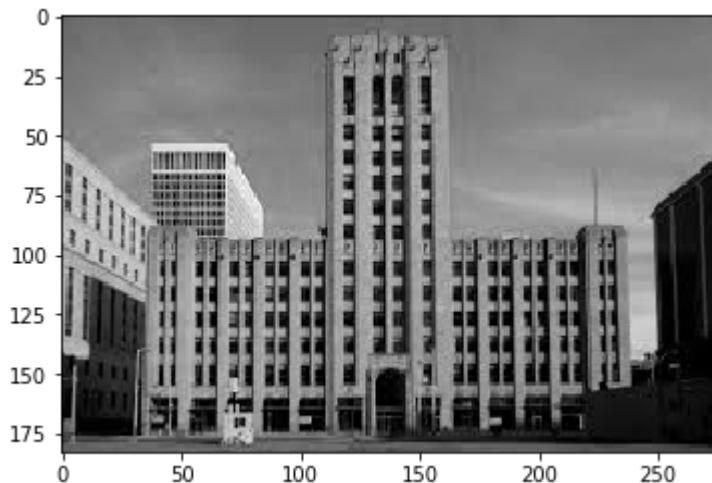
```
height, width, channels = image.shape  
image_grayscale = image.mean(axis=2).astype(np.float32)  
images = image_grayscale.reshape(1, height, width, 1)
```

In [27]:

```
plt.imshow(images[0,:,:,0], cmap = 'gray')
```

Out[27]:

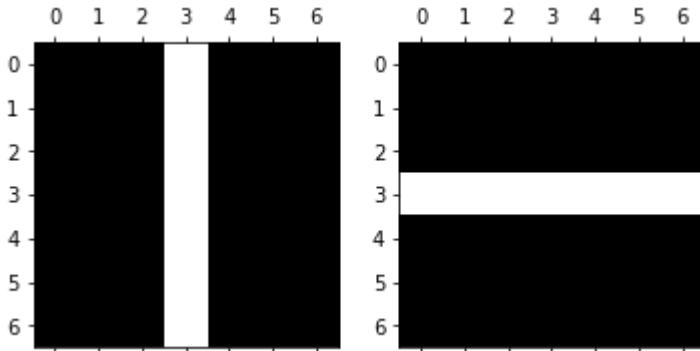
```
<matplotlib.image.AxesImage at 0x7f7fc7b47810>
```



Para tanto, vamos usar dois padrões: uma linha vertical e outra horizontal.

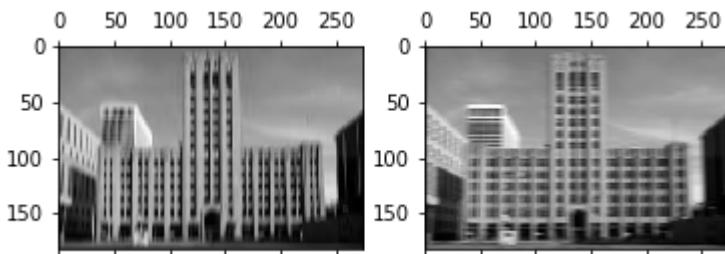
In [28]:

```
fmap = np.zeros(shape=(7, 7, 1, 2), dtype=np.float32)
fmap[:, 3, 0, 0] = 1
fmap[3, :, 0, 1] = 1
plot_figs([fmap[:, :, 0, 0], fmap[:, :, 0, 1]])
```



In [29]:

```
output = tfconv(images, fmap)
plot_figs([output[0, :, :, 0], output[0, :, :, 1]])
```



Percebam que, na imagem resultante, são ressaltadas as linhas verticais e horizontais da imagem original, enfatizando *onde* esses padrões são vistos.

A seguir, vamos parametrizar nossa função de convolução para observar o efeito de diferentes strides e padding.

In [30]:

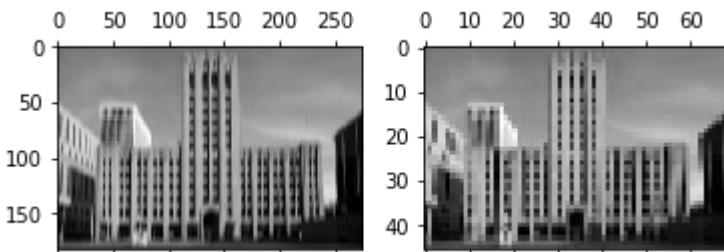
```
def tfconv(img, kernels, stride_val = 1, pad_val = 'SAME'):
    reset_graph()

    # batch, alt, larg, canais entrada
    X = tf.constant(img, dtype=tf.float32)
    feature_maps = tf.constant(kernels)
    convolution = tf.nn.conv2d(X, feature_maps,
                               strides=[1, stride_val, stride_val, 1],
                               padding=pad_val)

    with tf.Session() as s:
        output = convolution.eval(feed_dict={X: img})
    return output
```

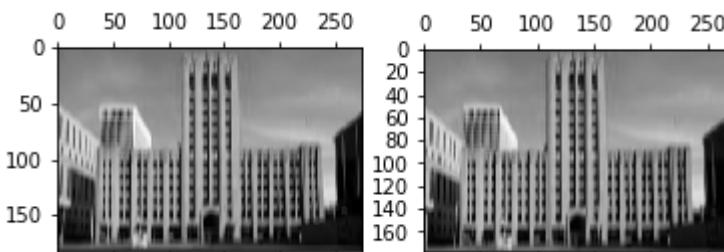
In [31]:

```
# stride 4 --> imagem é reduzida 4 vezes
output4 = tfconv(images, fmap, 4)
plot_figs([output[0, :, :, 0], output4[0, :, :, 0]])
```



In [32]:

```
# sem padding, quase não se nota mudança em uma imagem tão grande, exceto nas bordas.
outputv = tfconv(images, fmap, pad_val = 'VALID')
plot_figs([output[0, :, :, 0], outputv[0, :, :, 0]])
```



Vamos agora introduzir pooling no processo. Note que o efeito do pooling é similar a striding na convolução, ou seja, redução da imagem.

In [33]:

```
def tfconv_p(img, kernels, stride_val = 1, pad_val = 'SAME', pool_val = 2):
    reset_graph()

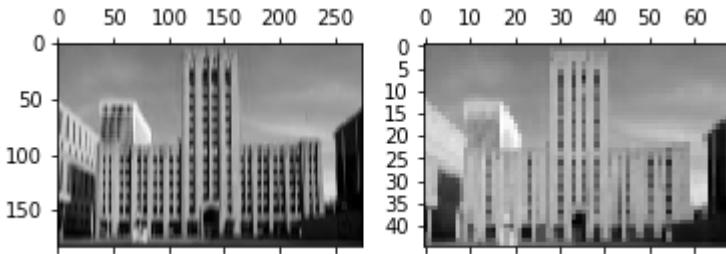
    # batch, alt, larg, canais entrada
    X = tf.constant(img, dtype=tf.float32)
    feature_maps = tf.constant(kernels)
    convolution = tf.nn.conv2d(X, feature_maps,
                              strides=[1, stride_val, stride_val, 1],
                              padding=pad_val)

    # max pool!!!
    max_pool = tf.nn.max_pool(convolution,
                             ksize=[1, pool_val, pool_val, 1],
                             strides=[1, pool_val, pool_val, 1],
                             padding="VALID")

    with tf.Session() as s:
        output = max_pool.eval(feed_dict={X: img})
    return output
```

In [34]:

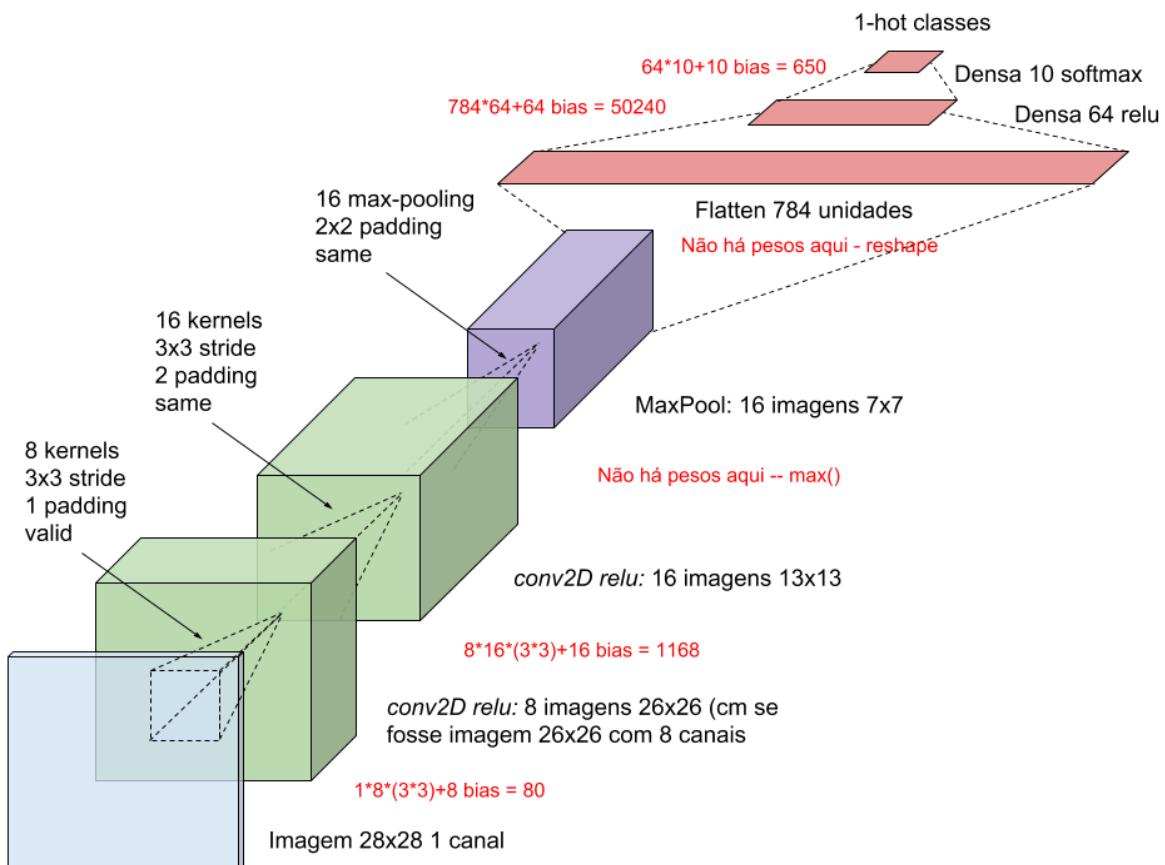
```
outputmax = tfconv_p(images, fmap, pool_val = 4)
plot_figs([output[0, :, :, 0], outputmax[0, :, :, 0]])
```



## Uma rede de convolução (CNN - Convolutional Neural Network)

Uma rede de convolução é basicamente uma rede neural com camadas em que se aplicam convoluções e pooling. Os pesos dos neurônios em camadas de convolução correspondem a kernels. Logo, os neurônios filtram as imagens que chegam em busca de padrões particulares. Os pesos nas camadas de pooling são sempre 1 e nunca se alteram. Ou seja, elas não fazem nada além de algum tipo de amostragem, reduzindo a quantidade de informação que vai seguir pela rede.

Abaixo, vamos usar uma CNN para classificar MNIST.



Nossa CNN tem  $28 \times 28$  sinais de entrada. Estes sinais são entregues para um rede de convolução com 8 kernels. As 8 imagens resultantes de  $28 \times 28$  pixels são entrada de uma segunda rede de convolução, com 16 kernels e stride 2. Ou seja, ela reduz a entrada produzindo 16 imagens de  $14 \times 14$  pixels. Esses passam por uma camada de pooling que os reduz para 16 imagens de  $7 \times 7$  pixels.

In [35]:

```
from keras import layers
from keras import models
import keras.backend as K

/home/marco/Enthought/Canopy_64bit/User/lib/python2.7/site-package
s/h5py/_init__.py:36: FutureWarning: Conversion of the second argu
ment of issubdtype from `float` to `np.floating` is deprecated. In
future, it will be treated as `np.float64 == np.dtype(float).type`.
    from ._conv import register_converters as _register_converters
Using TensorFlow backend.
```

In [36]:

```
K.clear_session()
```

In [37]:

```
model = models.Sequential()

model.add(layers.Conv2D(8, (3, 3), activation = 'relu', input_shape = (28, 28, 1
)))
model.add(layers.Conv2D(16, (3, 3), strides = (2, 2), padding = 'same', activati
on = 'relu'))
model.add(layers.MaxPooling2D((2, 2), padding = 'same'))
```

In [38]:

```
model.summary()
```

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 26, 26, 8)	80
conv2d_2 (Conv2D)	(None, 13, 13, 16)	1168
max_pooling2d_1 (MaxPooling2D)	(None, 7, 7, 16)	0
<hr/>		
Total params: 1,248		
Trainable params: 1,248		
Non-trainable params: 0		

Esses  $16 \times 7 \times 7$  sinais são então dados como entrada para 64 neurônios (camada que chamamos FC1) que, por sua vez, se conectam com outros 10 neurônios. Cada neurônio deste, então, representa um dígito diferente. Logo eles podem ser combinados via softmax para determinar o dígito mais provável da rede.

In [39]:

```
model.add(layers.Flatten())
model.add(layers.Dense(64, activation = 'relu', name = 'FC1'))
model.add(layers.Dense(10, activation = 'softmax', name = 'output'))
```

In [40]:

```
model.summary()
```

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 26, 26, 8)	80
conv2d_2 (Conv2D)	(None, 13, 13, 16)	1168
max_pooling2d_1 (MaxPooling2D)	(None, 7, 7, 16)	0
flatten_1 (Flatten)	(None, 784)	0
FC1 (Dense)	(None, 64)	50240
output (Dense)	(None, 10)	650

Total params: 52,138  
Trainable params: 52,138  
Non-trainable params: 0

In [41]:

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("data/MNIST_data")
```

Extracting data/MNIST\_data/train-images-idx3-ubyte.gz  
Extracting data/MNIST\_data/train-labels-idx1-ubyte.gz  
Extracting data/MNIST\_data/t10k-images-idx3-ubyte.gz  
Extracting data/MNIST\_data/t10k-labels-idx1-ubyte.gz

In [42]:

```
train_images = mnist.train.images.reshape((55000, 28, 28, 1))
test_images = mnist.test.images.reshape((10000, 28, 28, 1))
```

In [43]:

```
from keras.utils import to_categorical
```

In [44]:

```
train_labels = to_categorical(mnist.train.labels)
test_labels = to_categorical(mnist.test.labels)
```

In [45]:

```
model.compile(optimizer = 'adam', loss = 'categorical_crossentropy', metrics = ['accuracy'])
```

In [46]:

```
model.fit(train_images, train_labels, epochs = 5, batch_size = 64)
```

Epoch 1/5

```
55000/55000 [=====] - 4s - loss: 0.3084 -  
acc: 0.9076
```

Epoch 2/5

```
55000/55000 [=====] - 3s - loss: 0.1020 -  
acc: 0.9690
```

Epoch 3/5

```
55000/55000 [=====] - 3s - loss: 0.0723 -  
acc: 0.9777
```

Epoch 4/5

```
55000/55000 [=====] - 3s - loss: 0.0550 -  
acc: 0.9831
```

Epoch 5/5

```
55000/55000 [=====] - 3s - loss: 0.0439 -  
acc: 0.9861
```

Out[46]:

```
<keras.callbacks.History at 0x7f7efc18f090>
```

In [47]:

```
test_loss, test_acc = model.evaluate(test_images, test_labels)  
test_acc
```

```
9728/10000 [=====.>] - ETA: 0s
```

Out[47]:

```
0.9837
```

E a nossa primeira e simples CNN já atingiu uma alta taxa de classificação, da ordem de 98.5%.

Como seria o equivalente em tensorflow?

```

#
# modelo
#

height = 28
width = 28
channels = 1
n_inputs = height * width

reset_graph()

with tf.name_scope("inputs"):
    X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
    X_reshaped = tf.reshape(X, shape=(-1, height, width, channels))
    y = tf.placeholder(tf.int32, shape=(None), name="y")

# input: 28x28 image (batch images)
conv1 = tf.layers.conv2d(X_reshaped, filters=8, kernel_size=3,
                       strides=1, padding='SAME',
                       activation=tf.nn.relu, name="conv1")
# input: 8 28x28
conv2 = tf.layers.conv2d(conv1, filters=16, kernel_size=3,
                       strides=2, padding='SAME',
                       activation=tf.nn.relu, name="conv2")

with tf.name_scope("pool3"):
    # input: 16 14x14
    pool3 = tf.nn.max_pool(conv2, ksize=[1, 2, 2, 1],
                           strides=[1, 2, 2, 1], padding="VALID")
    # output: 16 7x7
    pool3_flat = tf.reshape(pool3, shape=(-1, 16 * 7 * 7))

with tf.name_scope("fc1"):
    # input: 16x7x7
    fc1 = tf.layers.dense(pool3_flat, 64, activation=tf.nn.relu, name="fc1")

with tf.name_scope("output"):
    logits = tf.layers.dense(fc1, 10, name="output")
    Y_proba = tf.nn.softmax(logits, name="Y_proba")

with tf.name_scope("train"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=logits, labels=y)
    loss = tf.reduce_mean(xentropy)

```

A seguir, vamos tentar melhorar nossa arquitetura introduzindo dropout nas camadas de pooling e FC1, além de permitir que ela treine por mais épocas (o treino será finalizado usando *early stopping*).

In [48]:

```
K.clear_session()
```

In [49]:

```
# com dropout
model = models.Sequential()

model.add(layers.Conv2D(8, (3, 3), activation = 'relu', input_shape = (28, 28, 1)))
model.add(layers.Conv2D(16, (3, 3), strides = (2, 2), padding = 'same', activation = 'relu'))
model.add(layers.MaxPooling2D((2, 2), padding = 'same'))

model.add(layers.Flatten())
model.add(layers.Dropout(0.25))
model.add(layers.Dense(64, activation = 'relu', name = 'FC1'))
model.add(layers.Dropout(0.50))
model.add(layers.Dense(10, activation = 'softmax', name = 'output'))

model.compile(optimizer = 'adam', loss = 'categorical_crossentropy', metrics = ['accuracy'])
```

In [50]:

```
model.fit(train_images, train_labels, epochs = 5, batch_size = 64)
```

```
Epoch 1/5
55000/55000 [=====] - 4s - loss: 0.5602 -
acc: 0.8216
Epoch 2/5
55000/55000 [=====] - 4s - loss: 0.2473 -
acc: 0.9255
Epoch 3/5
55000/55000 [=====] - 3s - loss: 0.1897 -
acc: 0.9420
Epoch 4/5
55000/55000 [=====] - 3s - loss: 0.1625 -
acc: 0.9514
Epoch 5/5
55000/55000 [=====] - 3s - loss: 0.1487 -
acc: 0.9555
```

Out[50]:

```
<keras.callbacks.History at 0x7f7e01123110>
```

In [51]:

```
test_loss, test_acc = model.evaluate(test_images, test_labels)
test_acc
```

```
9440/10000 [=====>..] - ETA: 0s
```

Out[51]:

```
0.9833
```

Para implementar *early stopping*, vamos usar um *callback* do Keras.

In [52]:

```
from keras.callbacks import EarlyStopping
```

In [53]:

```
K.clear_session()
```

In [54]:

```
# com dropout
model = models.Sequential()

model.add(layers.Conv2D(8, (3, 3), activation = 'relu', input_shape = (28, 28, 1)))
model.add(layers.Conv2D(16, (3, 3), strides = (2, 2), padding = 'same', activation = 'relu'))
model.add(layers.MaxPooling2D((2, 2), padding = 'same'))

model.add(layers.Flatten())
model.add(layers.Dropout(0.25))
model.add(layers.Dense(64, activation = 'relu', name = 'FC1'))
model.add(layers.Dropout(0.50))
model.add(layers.Dense(10, activation = 'softmax', name = 'output'))

earlystopping = EarlyStopping(monitor='val_loss', patience = 2, min_delta = 0, verbose=0)

model.compile(optimizer = 'adam', loss = 'categorical_crossentropy', metrics = ['accuracy'])
```

In [55]:

```
model.fit(train_images, train_labels, epochs = 20, validation_split = 0.05,
          batch_size = 64, callbacks = [earlystopping])
```

Train on 52250 samples, validate on 2750 samples  
Epoch 1/20  
52250/52250 [=====] - 4s - loss: 0.5368 -  
acc: 0.8305 - val\_loss: 0.0914 - val\_acc: 0.9778  
Epoch 2/20  
52250/52250 [=====] - 3s - loss: 0.2156 -  
acc: 0.9358 - val\_loss: 0.0615 - val\_acc: 0.9858  
Epoch 3/20  
52250/52250 [=====] - 3s - loss: 0.1754 -  
acc: 0.9489 - val\_loss: 0.0543 - val\_acc: 0.9873  
Epoch 4/20  
52250/52250 [=====] - 4s - loss: 0.1483 -  
acc: 0.9554 - val\_loss: 0.0474 - val\_acc: 0.9880  
Epoch 5/20  
52250/52250 [=====] - 3s - loss: 0.1359 -  
acc: 0.9585 - val\_loss: 0.0456 - val\_acc: 0.9880  
Epoch 6/20  
52250/52250 [=====] - 3s - loss: 0.1242 -  
acc: 0.9622 - val\_loss: 0.0452 - val\_acc: 0.9876  
Epoch 7/20  
52250/52250 [=====] - 3s - loss: 0.1172 -  
acc: 0.9640 - val\_loss: 0.0464 - val\_acc: 0.9891  
Epoch 8/20  
52250/52250 [=====] - 3s - loss: 0.1138 -  
acc: 0.9651 - val\_loss: 0.0421 - val\_acc: 0.9905  
Epoch 9/20  
52250/52250 [=====] - 4s - loss: 0.1078 -  
acc: 0.9655 - val\_loss: 0.0416 - val\_acc: 0.9913  
Epoch 10/20  
52250/52250 [=====] - 3s - loss: 0.1078 -  
acc: 0.9677 - val\_loss: 0.0377 - val\_acc: 0.9902  
Epoch 11/20  
52250/52250 [=====] - 3s - loss: 0.1032 -  
acc: 0.9694 - val\_loss: 0.0347 - val\_acc: 0.9902  
Epoch 12/20  
52250/52250 [=====] - 3s - loss: 0.0986 -  
acc: 0.9698 - val\_loss: 0.0383 - val\_acc: 0.9898  
Epoch 13/20  
52250/52250 [=====] - 3s - loss: 0.0973 -  
acc: 0.9699 - val\_loss: 0.0352 - val\_acc: 0.9916  
Epoch 14/20  
52250/52250 [=====] - 3s - loss: 0.0912 -  
acc: 0.9714 - val\_loss: 0.0375 - val\_acc: 0.9902

Out[55]:

```
<keras.callbacks.History at 0x7f7e00c9d850>
```

In [56]:

```
test_loss, test_acc = model.evaluate(test_images, test_labels)
test_acc
```

9312/10000 [=====>...] - ETA: 0s

Out[56]:

0.9877

Esta rede, mesmo usando poucos kernels já é capaz de alcançar taxas de acerto superiores a 99%.

Como isso poderia ser feito em tensorflow?

```

# save and restore net in memory
def get_model_params():
    gvars = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES)
    return {gvar.op.name: value for gvar, value in zip(gvars, tf.get_default_session().run(gvars))}

def restore_model_params(model_params):
    gvar_names = list(model_params.keys())
    assign_ops = {gvar_name: tf.get_default_graph().get_operation_by_name(gvar_name + "/Assign")
                 for gvar_name in gvar_names}
    init_values = {gvar_name: assign_op.inputs[1] for gvar_name, assign_op in assign_ops.items()}
    feed_dict = {init_values[gvar_name]: model_params[gvar_name] for gvar_name in gvar_names}
    tf.get_default_session().run(assign_ops, feed_dict=feed_dict)

# treino com early stopping
n_epochs = 1000
batch_size = 50

best_loss_val = np.infty
check_interval = 500
no_progress = 0
max_no_progress = 20
best_model_params = None

with tf.Session() as s:
    init.run()
    for e in range(n_epochs):
        for i in range(mnist.train.num_examples // batch_size):
            X_b, y_b = mnist.train.next_batch(batch_size)
            s.run(training_op, feed_dict={X: X_b, y: y_b, training: True})
    # dropout
        if i % check_interval == 0:
            loss_val = loss.eval(feed_dict={X: mnist.validation.images,
                                             y: mnist.validation.labels})
            if loss_val < best_loss_val:
                best_loss_val = loss_val
                no_progress = 0
                best_model_params = get_model_params()
            else:
                no_progress += 1
            acc_train = accuracy.eval(feed_dict={X: X_b, y: y_b})
            acc_val = accuracy.eval(feed_dict={X: mnist.validation.images,
                                              y: mnist.validation.labels})
            print("%2d: train acc: %.5f, val acc: %.5f, curr best loss: %.5f"
                  % (e, acc_train, acc_val, best_loss_val))
    .....
```

## Observando pesos de camadas

Uma forma de compreender melhor redes neurais é observando os pesos que foram aprendidos. No caso de processamento de imagens, onde os pesos representam os padrões visuais que os neurônios reconhecem, eles podem ser facilmente interpretáveis.

Abaixo listamos os parâmetros observados no melhor modelo aprendido anteriormente.

In [57]:

```
weights = model.get_weights()
```

In [58]:

```
wnames = []
for l in model.layers:
    for ws in l.weights:
        wnames += [ws.name]
wnames
```

Out[58]:

```
[u'conv2d_1/kernel:0',
 u'conv2d_1/bias:0',
 u'conv2d_2/kernel:0',
 u'conv2d_2/bias:0',
 u'FC1/kernel:0',
 u'FC1/bias:0',
 u'output/kernel:0',
 u'output/bias:0']
```

In [59]:

```
dweights = dict(zip(wnames, weights))
```

In [63]:

```
dweights['conv2d_1/kernel:0'].shape
```

Out[63]:

```
(3, 3, 1, 8)
```

In [64]:

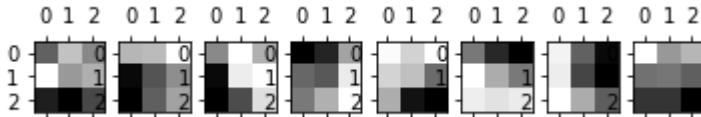
```
dweights['conv2d_1/kernel:0'][[:, :, 0, 0].reshape(3, 3)]
```

Out[64]:

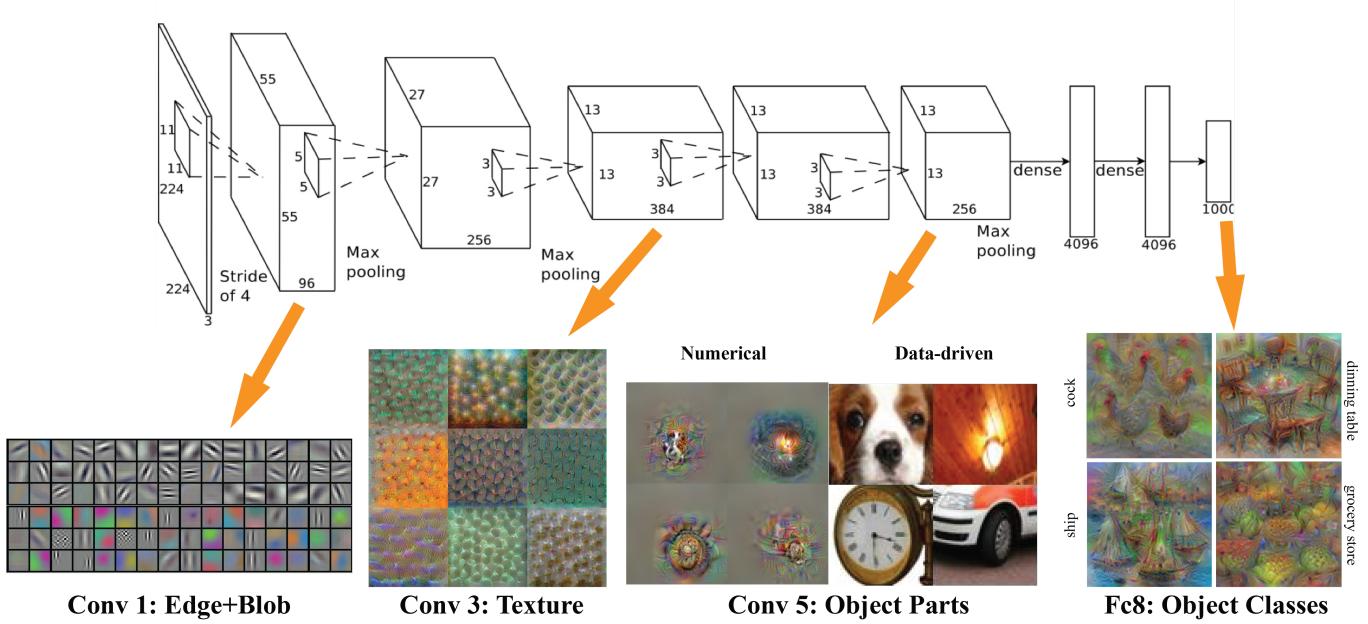
```
array([[-0.11671568,  0.28047413,  0.04168296],
       [ 0.5360935 ,  0.11273681,  0.20200661],
       [-0.39730203, -0.5249813 , -0.2225567 ]], dtype=float32)
```

In [66]:

```
plot_figs([dweights['conv2d_1/kernel:0'][::,:,0,i] for i in range(8)])
```



Note que como nesta rede usamos kernels de tamanho  $3 \times 3$ , não há padrões muito significativos a observar. Este não é o caso de redes onde são usados kernels maiores, em que padrões muito característicos são observados. Por exemplo, abaixo, podemos ver kernels de diferentes camadas na rede AlexNet (imagem de [http://vision03.csail.mit.edu/cnn\\_art/index.html](http://vision03.csail.mit.edu/cnn_art/index.html) ([http://vision03.csail.mit.edu/cnn\\_art/index.html](http://vision03.csail.mit.edu/cnn_art/index.html))):



Claramente, camadas mais profundas apresentam padrões mais complexos. Assim, vemos bordas e manchas coloridas na primeira camada, texturas na terceira, partes de objetos na quinta e padrões complexos na oitava.

## Rede Neural como codificador distribuído

Outra característica interessante de redes neurais é que a saída dela para uma instância em qualquer camada, pode ser tomada como uma representação *comprimida* e *distribuída* daquela instância (um *embedding*). Como tais representações são baseadas nos mesmos conjuntos de características, espera-se que elas sejam similares para instâncias que compartilham as mesmas características. Logo, duas instâncias do número 0 devem ter representação similar, porém diferente do número 1, por exemplo.

Para observarmos isso, vamos tomar a representação correspondente à camada totalmente conectada da nossa rede, a fc1:

In [69]:

```
dweights['FC1/kernel:0'].shape
```

Out[69]:

(784, 64)

In [70]:

```
from keras.models import Model
```

In [71]:

```
model_at_layer = Model(inputs=model.input,
                       outputs=model.get_layer('FC1').output)
Xs = mnist.test.images[:100].reshape(100,28,28,1)
fc1_vals = model_at_layer.predict(Xs)
```

Note que fc1\_vals tem as saídas dos neurônios da camada fc1 para as 100 primeiras imagens da coleção de teste.

In [72]:

```
from sklearn.neighbors import NearestNeighbors
import random
```

In [73]:

```
neigh = NearestNeighbors(2)
neigh.fit(fc1_vals)
```

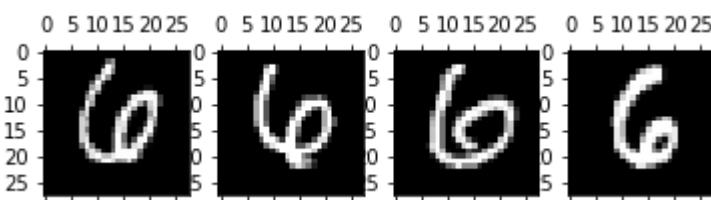
Out[73]:

```
NearestNeighbors(algorithm='auto', leaf_size=30, metric='minkowski',
                  metric_params=None, n_jobs=1, n_neighbors=2, p=2, radius=
1.0)
```

Se tomarmos aleatoriamente uma imagem em fc1\_vals e seus 4 vizinhos mais próximos, de acordo com a representação feita pelos neurônios, observe o que obtemos:

In [74]:

```
rquery = random.randint(0, 99)
indices = neigh.kneighbors([fc1_vals[rquery]], 4, return_distance=False)
plot_figs([mnist.test.images[i].reshape(28,28) for i in indices[0]])
```



## Camadas Inception

As primeiras redes de convolução, como a AlexNet, usavam grandes kernels (ex: 14x14). Enquanto isso é interessante para visualizarmos o que a rede aprendeu, por outro lado, implica em fazer muitas operações, a saber, número de mapas na entrada  $\times$  altura da entrada  $\times$  largura da entrada  $\times$  (tamanho do kernel  $\times$  tamanho do kernel)  $\times$  número de mapas na saída.

Por exemplo, se a camada de convolução vai processar uma entrada de 28x28 com 192 mapas de 14x14 para entregar para uma camada seguinte que possui 32 mapas, ela vai consumir  
 $14^2 * 28^2 * 192 * 32 = 944.111.616$  operações. Muito!

Logo, parece mais aconselhável usar kernels menores. Mas que tamanho? 3x3? 5x5? Uma solução é usar ambos e, então, deixar para a camada da frente da rede neural decidir o que fazer. Esta ideia foi inspirada em uma arquitetura chamada NiN (2014) e deu origem à Inception v1.

Embora esta nova abordagem permitisse à rede explorar padrões maiores e menores no mesmo campo de visão, ela continuava a requerer muitas operações. Em particular,  
 $(5^2 * 28^2 * 192 * 32 + 3^2 * 28^2 * 192 * 32) = 163.774.464$ . Embora bem menos que nas primeiras arquiteturas, ainda é muito.

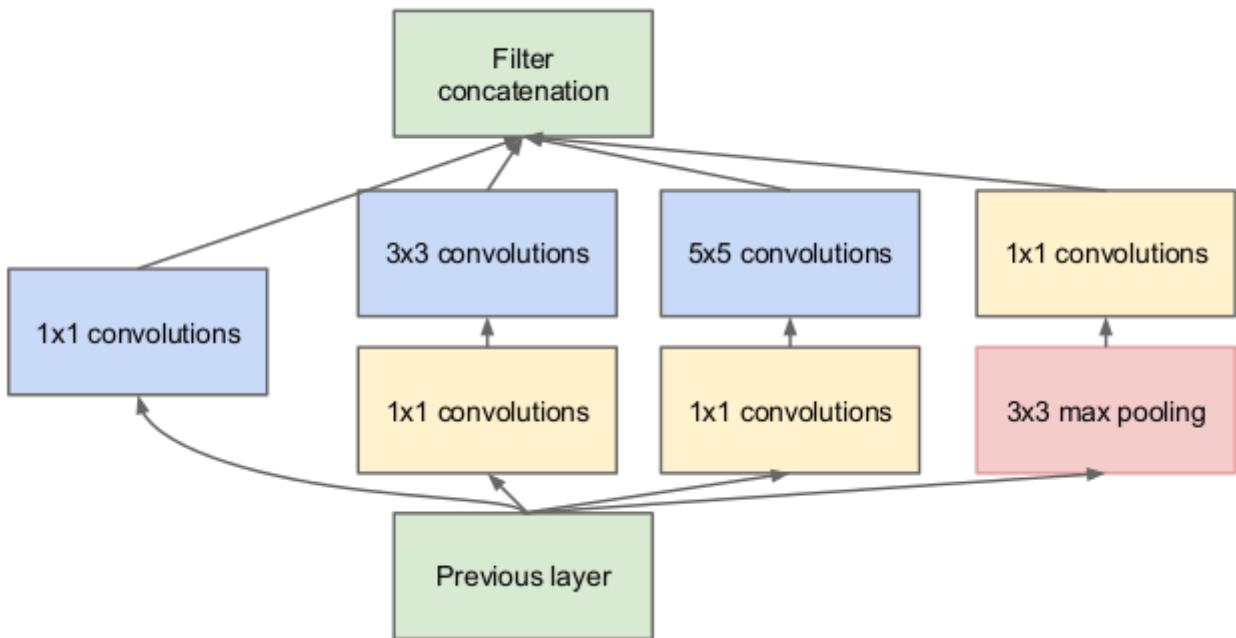
Uma solução proposta foi usar um número menor de kernels intermediários de tamanho 1x1, funcionando como combinadores lineares.

Por que kernels 1x1 são melhores? Porque se usarmos um número menor de kernels 1x1 (ex: 16) entre os 192 e os 32 kernels usados antes, o número global de operações será menor. Ou seja, em lugar de 192 kernels 3x3 e 5x5 processando entradas de 28x28 para 32 kernels de saída, teremos agora 192 kernels 3x3 e 5x5 processando entradas de 28x28 para 16 kernels 1x1. Estes, por sua vez, serão entrada de 32 kernels de saída. O número de operações é, então:

$$(1^2 * 28^2 * 192 * 32 + 5^2 * 28^2 * 16 * 32) + (1^2 * 28^2 * 192 * 32 + 3^2 * 28^2 * 16 * 32) = 23.281.664$$

Muito menos!!!

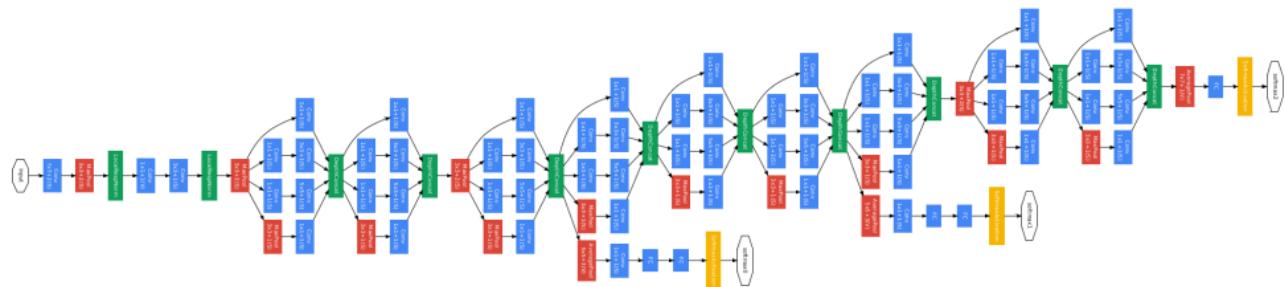
Redes de convolução Inception v2 basicamente vão incorporar essa ideia. Uma camada inception, em particular, é compostas por 4 operações paralelas: (1) convolução 1x1, (2) convolução 1x1 em série com 3x3, (3) convolução 1x1 em série com 5x5 e (4) uma maxpooling (pq? bom, pq todo mundo usa maxpooling) em série com convolução 1x1. Veja abaixo:



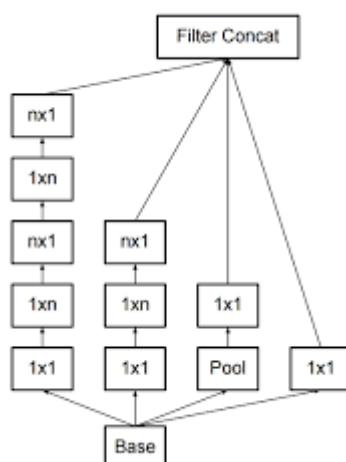
Ao reduzir o número de operações & parâmetros, foi possível conseguir redes mais profundas. Com redes mais profundas, contudo, o problema de perda de gradientes se agravou, o que levou à ideia de fazer supervisão em várias camadas intermediárias da rede (em lugar de apenas no fim). Gradientes intermediários são então somados aos gradientes globais da rede durante a propagação retrógrada de forma a minimizar o problema de perda de gradientes.

Abaixo você pode ver 3 camadas softmax na rede GoogLeNet representadas por blocos laranja, além de nove camadas inception (fonte:

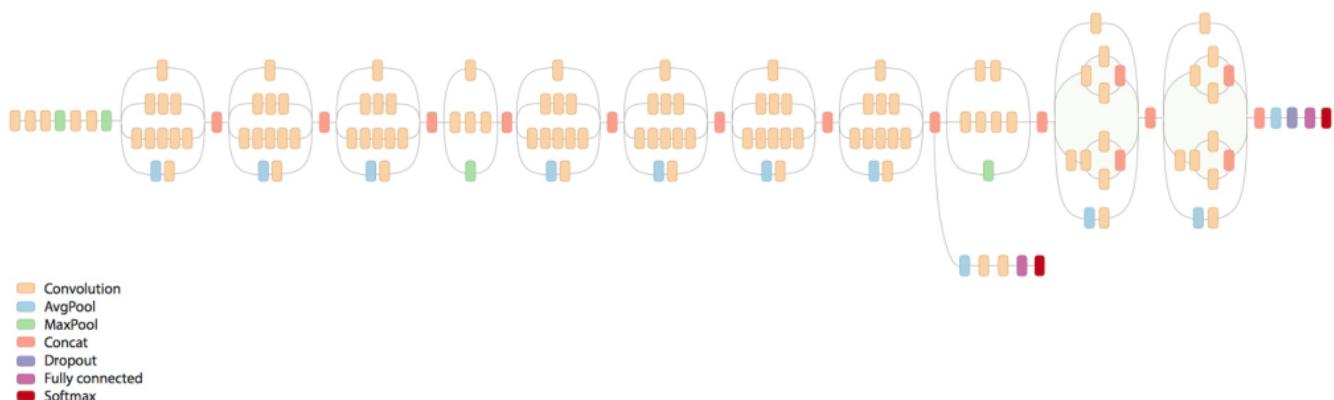
[http://joelouismarino.github.io/images/blog\\_images/blog\\_googlenet\\_keras/googlenet\\_diagram.png](http://joelouismarino.github.io/images/blog_images/blog_googlenet_keras/googlenet_diagram.png)  
 ([http://joelouismarino.github.io/images/blog\\_images/blog\\_googlenet\\_keras/googlenet\\_diagram.png](http://joelouismarino.github.io/images/blog_images/blog_googlenet_keras/googlenet_diagram.png))).



A evolução das redes Inception seguiu com a introdução de BN (ainda na Inception v2) e uso de kernels de no máximo tamanho 3 (kernels 3x3). Para cobrir áreas tão amplas quanto a dos kernels 5x5 e 7x7 adotou-se o empilhamento de kernels 3x3 e/ou séries de kernels  $1 \times n$  seguidos de  $n \times 1$  ( $n = 7$  em Inception v3).



A arquitetura baseada nesta nova camada é a Inception v3 (ver <https://arxiv.org/pdf/1512.00567.pdf> (<https://arxiv.org/pdf/1512.00567.pdf>) para uma descrição de todas elas):



Arquiteturas Inception seguintes (v4) incluem camadas residuais, discutidas a seguir.

# Redes de Convolução Residuais

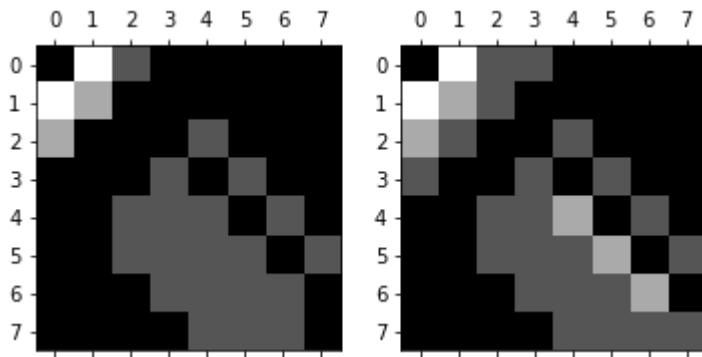
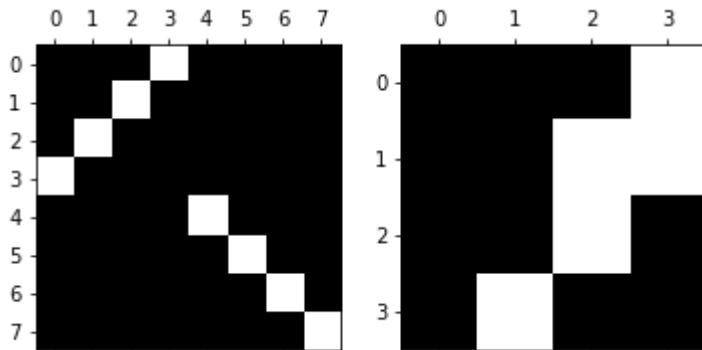
Observe as operações a seguir:

In [77]:

```
k3x3 = np.array([[0,0,0,1],  
                 [0,0,1,1],  
                 [0,0,1,0],  
                 [0,1,0,0]], dtype=np.float32)
```

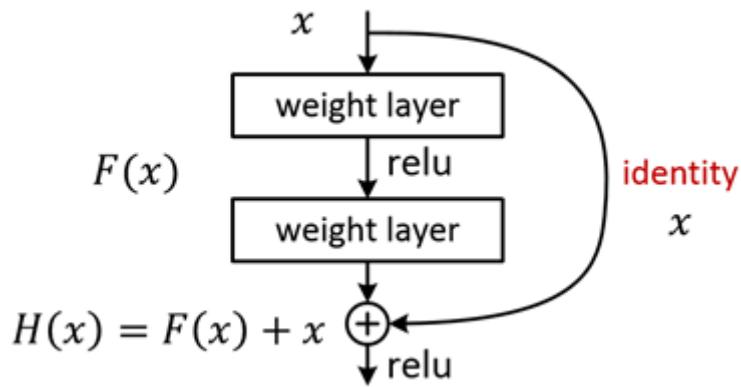
In [78]:

```
fmap3x3 = k3x3.reshape(k3x3.shape[0], k3x3.shape[1], 1, 1)  
input0 = ex0.reshape(1, ex0.shape[0], ex0.shape[1], 1)  
output0 = tfconv(input0, fmap3x3)  
plot_figs([ex0, k3x3])  
plot_figs([output0[0, :, :, 0], ex0 + output0[0, :, :, 0]])
```



Como antes, o kernel 3x3 foi aplicado à imagem ex0, produzindo uma interpretação onde o kernel parece observado na imagem. A quarta imagem corresponde à soma da entrada com a saída -- dizemos que a entrada foi 'copiada' para a saída. Ou seja, temos uma mistura entre onde o kernel foi observado e sua entrada original ∴. Como interpretar isso?

Note que uma pilha de camadas de uma rede neural aprende uma representação/mapeamento  $\mathcal{F}(\mathbf{x})$  para a sua entrada  $\mathbf{x}$ . Ao somarmos  $\mathbf{x}$  à saída  $\mathcal{F}$  da camada, o código aprendido passa a ser  $\mathcal{H}(\mathbf{x}) = \mathcal{F}(\mathbf{x}) + \mathbf{x}$ , ou seja, a rede aprende o resíduo da representação  $\mathcal{F}(\mathbf{x}) = \mathcal{H}(\mathbf{x}) - \mathbf{x}$ .



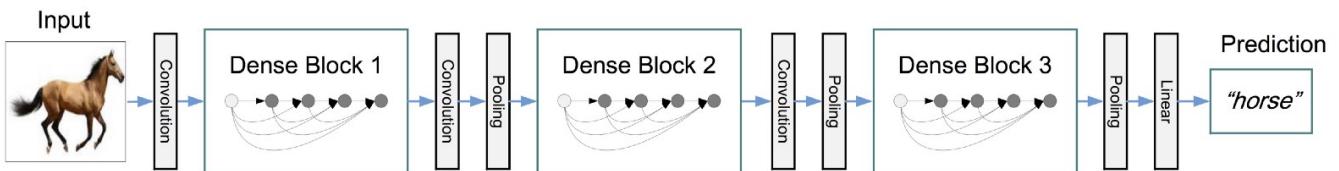
Mas por que isso seria útil? Em termos teóricos, se um mapeamento existente  $\mathbf{x}$  já é ótimo, é mais simples uma pilha de camadas aprender um resíduo  $\mathcal{F}(\mathbf{x})$  próximo a zero que o mapeamento da função identidade através de uma pilha de camadas não lineares. Generalizando esta ideia, podemos dizer que é mais simples otimizar resíduos.

Em termos práticos, os neurônios que ainda não aprenderam nada útil e estão disparando próximo a zero, agora disparam uma cópia do valor da sua entrada (ou de uma entrada anterior, dependendo de onde é feita a 'cópia'). Assim, eles não bloqueiam mais o fluxo do sinal pela rede através das camadas, o que nos permite usar mais camadas!!! O resultado é a criação de redes com centenas de camadas, sem a necessidade de treinamento intermediário.

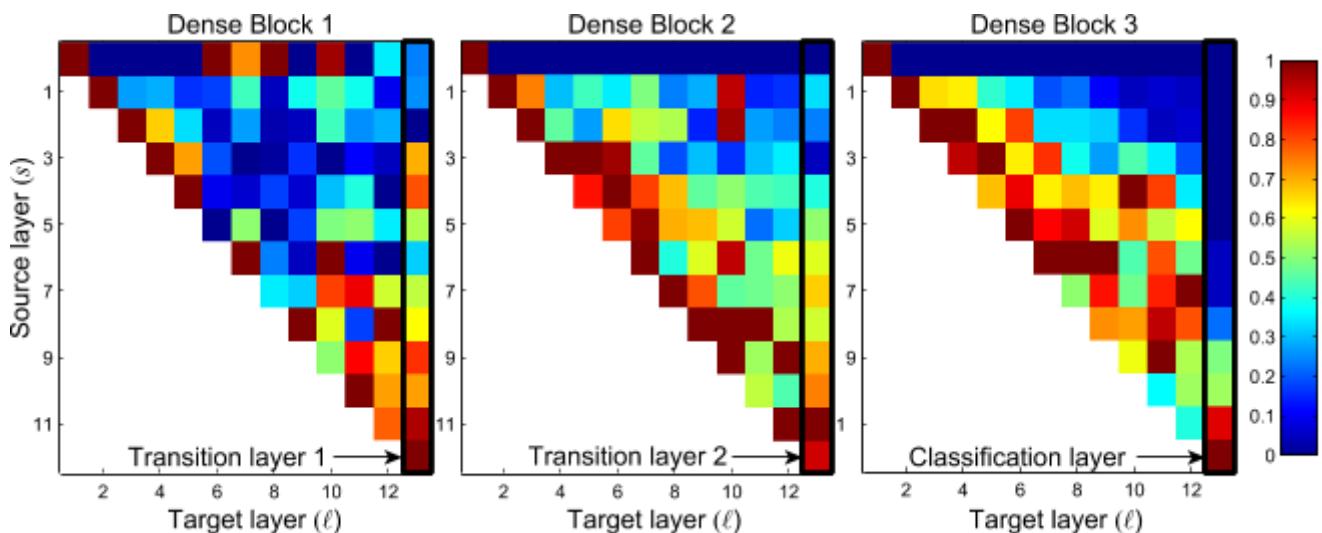
## Redes de Convolução Densas

Tanto redes inception quanto residuais sugerem que camadas podem tirar proveito de atributos menos abstratos (obtidos via kernels menores em Inception ou de camadas anteriores em ResNets).

Várias arquiteturas exploraram esta ideia de que atributos aprendidos em camadas anteriores podem ser úteis à frente, como as redes Fractais (2016). A ideia, contudo, foi generalizada pelas redes de convolução densamente conectadas (<https://arxiv.org/pdf/1608.06993.pdf>). Nestas, qualquer camada tem acesso a todos os atributos aprendidos em todas as camadas anteriores que seguem uma camada de pooling (uma vez que camadas de pooling mudam o tamanho do atributo -- kernel, o que impediria a sua utilização).



O estudo destas arquiteturas demonstra que camadas posteriores tem alta probabilidade de usar atributos de camadas anteriores próximas. Isso é mais relevante quanto mais perto do início da rede está a camada, como visto na figura a seguir do paper sobre *densely connected convolution nets* (onde cores indicam a influência das camadas).



In [ ]: