In [1]:
```
%matplotlib inline
```

In [2]:
```
import tensorflow as tf
import tensorflow.contrib.slim as slim
from scipy.io import loadmat
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
import os
import itertools
import math
from ipywidgets import interact, interactive, fixed
import ipywidgets as widgets
```

In [3]:
```
import classifier_utils as utils
```

Get the training data

In [4]:
```
categories = [
    'epithelial',
    'fibroblast',
    'inflammatory',
    'others',
]
```

In [5]:
```
# Read in the raw data
(raw_imgs, raw_centres, raw_labels) = utils.get_dataset(100, categories)
```

In [6]:
```
# Extract example patches from the data
H = 27
W = 27
(patches, labels, centres, img_ids) = utils.get_examples(
    raw_imgs, raw_centres, raw_labels, H, W)
```

```
Dropped 2082 patches because too close to image border
```

In [7]:
```
# Organize examples into training and test data
N = patches.shape[0]
num_train = int(0.8 * N)

np.random.seed(0) # predictable shuffling for now
perm = np.random.permutation(N)

train_patches, test_patches = np.split(patches[perm], [num_train])
train_labels, test_labels = np.split(labels[perm], [num_train])
train_centres, test_centres = np.split(centres[perm], [num_train])
train_img_ids, test_img_ids = np.split(img_ids[perm], [num_train])

# Convert to float
train_patches = train_patches / 255.0
test_patches = test_patches / 255.0
```

In [8]:
```
# Sanity check
train_patches.shape, train_labels.shape, test_patches.shape,
test_labels.shape
```

Out[8]:
```
((16289, 27, 27, 3), (16289, 4), (4073, 27, 27, 3), (4073, 4))
```

In [9]:
```
# Expand/augment the training data
desired_cnt_per_category = 15000
sorted_train_dict = utils.expand_training_data(
    raw_imgs, train_patches, train_labels, train_centres, train_img_ids,
desired_cnt_per_category)
# Convert to float
sorted_train_dict['patches'] = sorted_train_dict['patches'] / 255.0
```

In [10]:
```
# Suffle the augmented training data
trainN = sorted_train_dict['patches'].shape[0]

np.random.seed(123) # predictable shuffling for now
perm = np.random.permutation(trainN)
train_dict = {k : v[perm] for (k, v) in sorted_train_dict.iteritems()}
```

In [11]:
```
sess = tf.InteractiveSession()
```

In [12]:
```
def upscale(images, scales, name):
    (yscale, xscale) = scales
    (batch_size, height, width, channels) = images.get_shape().as_list()
    with tf.name_scope(name):
        return tf.image.resize_images(images, height*yscale,
width*xscale,
                                      method=tf.image.ResizeMethod.NEARES
T_NEIGHBOR)
```

```
def autoencoder_model_1(image_batch):
    with slim.arg_scope([slim.conv2d, slim.fully_connected],
                        activation_fn=tf.nn.relu,
                        weights_initializer=tf.truncated_normal_initializ
er(stddev=0.01),
                        biases_initializer=tf.zeros_initializer,
                        #weights_regularizer=slim.l2_regularizer(0.0005),
                        biases_regularizer=None):
        with slim.arg_scope([slim.conv2d, slim.conv2d_transpose],
                            padding='VALID'):
            with slim.arg_scope([slim.dropout], keep_prob=0.8):
                with slim.arg_scope([slim.conv2d, slim.fully_connected, s
lim.conv2d_transpose],
                                    normalizer_fn=slim.batch_norm):
                    net = image_batch

                    estack = []
                    def r(x):
                        estack.append(x)
                        return x
                    with tf.variable_scope("encoder"):
                        net = r(slim.conv2d(net, 36, [4, 4], scope='1_con
v'))
                        net = r(slim.conv2d(net, 42, [3, 3], scope='2_con
v'))
                        net = r(slim.max_pool2d(net, [2, 2], scope='3_max
_pool'))
                        net = r(slim.conv2d(net, 48, [4, 4], scope='4_con
v'))
                        net = r(slim.max_pool2d(net, [2, 2], scope='5_max
_pool'))
                        net = r(slim.flatten(net, scope='5_flatten'))
                        net = r(slim.fully_connected(net, 256, scope='6_f
c'))
                        #net = r(slim.dropout(net, scope='6_dropout'))
                        net = r(slim.fully_connected(net, 64, scope='7_f
c'))

                    encoded = net

                    dstack = []
                    def r(x):
                        dstack.append(x)
                        return x
                    with tf.variable_scope("decoder"):
                        net = r(slim.fully_connected(net, 256, scope='1_f
c'))
                        #net = r(slim.dropout(net, scope='1_dropout'))
                        net = r(slim.fully_connected(net, 4*4*48,
scope='2_fc'))
                        net = r(tf.reshape(net, [-1, 4, 4, 48], name='2_r
eshape'))
                        net = r(upscale(net, [2, 2], name='3_upscale'))
                        net = r(slim.conv2d_transpose(net, 42, [4, 4], sc
ope='4_conv'))
                        net = r(upscale(net, [2, 2], name='5_upscale'))
                        net = r(slim.conv2d_transpose(net, 36, [3, 3], sc
```

```
ope='6_conv'))
                    net = r(slim.conv2d_transpose(net, 3, [4, 4], sco
pe='7_conv'))
                reconstructed = net
                return (encoded, reconstructed, estack, dstack)
```

```
def autoencoder_model_2(image_batch):
    with slim.arg_scope([slim.conv2d, slim.fully_connected],
                        activation_fn=tf.nn.relu,
                        weights_initializer=tf.truncated_normal_initializ
er(stddev=0.01),
                        biases_initializer=tf.zeros_initializer,
                        #weights_regularizer=slim.l2_regularizer(0.0005),
                        biases_regularizer=None):
        with slim.arg_scope([slim.conv2d, slim.conv2d_transpose],
                            padding='VALID'):
            with slim.arg_scope([slim.dropout], keep_prob=0.8):
                with slim.arg_scope([slim.conv2d, slim.fully_connected, s
lim.conv2d_transpose],
                                    normalizer_fn=slim.batch_norm):
                    net = image_batch

                    estack = []
                    def r(x):
                        estack.append(x)
                        return x
                    with tf.variable_scope("encoder"):
                        net = r(slim.conv2d(net, 36, [4, 4], scope='1_con
v'))
                        net = r(slim.conv2d(net, 42, [3, 3], scope='2_con
v'))
                        net = r(slim.max_pool2d(net, [2, 2], scope='3_max
_pool'))
                        net = r(slim.conv2d(net, 48, [4, 4], scope='4_con
v'))
                        net = r(slim.max_pool2d(net, [2, 2], scope='5_max
_pool'))
                        net = r(slim.flatten(net, scope='5_flatten'))
                        net = r(slim.fully_connected(net, 256, scope='6_f
c'))
                        #net = r(slim.dropout(net, scope='6_dropout'))
                        net = r(slim.fully_connected(net, 128, scope='7_f
c'))

                    encoded = net

                    dstack = []
                    def r(x):
                        dstack.append(x)
                        return x
                    with tf.variable_scope("decoder"):
                        net = r(slim.fully_connected(net, 256, scope='1_f
c'))
                        #net = r(slim.dropout(net, scope='1_dropout'))
                        net = r(slim.fully_connected(net, 4*4*48,
scope='2_fc'))
                        net = r(tf.reshape(net, [-1, 4, 4, 48], name='2_r
eshape'))
                        net = r(upscale(net, [2, 2], name='3_upscale'))
                        net = r(slim.conv2d_transpose(net, 42, [4, 4], sc
ope='4_conv'))
                        net = r(upscale(net, [2, 2], name='5_upscale'))
                        net = r(slim.conv2d_transpose(net, 36, [3, 3], sc
```

```
ope='6_conv'))
                    net = r(slim.conv2d_transpose(net, 3, [4, 4], sco
pe='7_conv'))
                reconstructed = net
                return (encoded, reconstructed, estack, dstack)
```

```
def autoencoder_model_3(image_batch):
    with slim.arg_scope([slim.conv2d, slim.fully_connected],
                        activation_fn=tf.nn.relu,
                        weights_initializer=tf.truncated_normal_initializ
er(stddev=0.01),
                        biases_initializer=tf.zeros_initializer,
                        #weights_regularizer=slim.l2_regularizer(0.0005),
                        biases_regularizer=None):
        with slim.arg_scope([slim.conv2d, slim.conv2d_transpose],
                            padding='VALID'):
            with slim.arg_scope([slim.dropout], keep_prob=0.8):
                with slim.arg_scope([slim.conv2d, slim.fully_connected, s
lim.conv2d_transpose],
                                    normalizer_fn=slim.batch_norm):
                    net = image_batch

                    estack = []
                    def r(x):
                        estack.append(x)
                        return x
                    with tf.variable_scope("encoder"):
                        net = r(slim.conv2d(net, 36, [4, 4], scope='1_con
v'))
                        net = r(slim.conv2d(net, 42, [3, 3], scope='2_con
v'))
                        net = r(slim.max_pool2d(net, [2, 2], scope='3_max
_pool'))
                        net = r(slim.conv2d(net, 48, [4, 4], scope='4_con
v'))
                        net = r(slim.max_pool2d(net, [2, 2], scope='5_max
_pool'))
                        net = r(slim.flatten(net, scope='5_flatten'))
                        net = r(slim.fully_connected(net, 256, scope='6_f
c'))
                        #net = r(slim.dropout(net, scope='6_dropout'))
                        net = r(slim.fully_connected(net, 128, scope='7_f
c'))

                    encoded = net

                    dstack = []
                    def r(x):
                        dstack.append(x)
                        return x
                    with tf.variable_scope("decoder"):
                        net = r(slim.fully_connected(net, 256, scope='1_f
c'))
                        #net = r(slim.dropout(net, scope='1_dropout'))
                        net = r(slim.fully_connected(net, 4*4*48,
scope='2_fc'))
                        net = r(tf.reshape(net, [-1, 4, 4, 48], name='2_r
eshape'))
                        net = r(upscale(net, [2, 2], name='3_upscale'))
                        net = r(slim.conv2d_transpose(net, 42, [4, 4], sc
ope='4_conv'))
                        net = r(upscale(net, [2, 2], name='5_upscale'))
                        net = r(slim.conv2d_transpose(net, 36, [3, 3], sc
```

```
ope='6_conv'))
                        net = r(slim.conv2d_transpose(net, 3, [4, 4], act
ivation_fn=tf.nn.sigmoid, scope='7_conv'))

                    reconstructed = net

                    return (encoded, reconstructed, estack, dstack)
```

In [16]:
```
if sess is not None:
    sess.close()
tf.reset_default_graph()
sess = tf.InteractiveSession()
patch_tensor = tf.placeholder(dtype='float32', shape=(None, 27, 27, 3))
with tf.variable_scope("autoencoder"):
    with slim.arg_scope([slim.dropout], is_training=True), slim.arg_scope
lim.conv2d, slim.conv2d_transpose, slim.fully_connected], normalizer_para
ms={'is_training':True}):
        (encoded_tensor, reconstructed_tensor, estack, dstack) = autoenco
der_model_3(patch_tensor)
with tf.variable_scope("autoencoder", reuse=True):
    with slim.arg_scope([slim.dropout], is_training=False), slim.arg_scop
e([slim.conv2d, slim.conv2d_transpose, slim.fully_connected], normalizer_
params={'is_training':False}):
        (eval_encoded_tensor, eval_reconstructed_tensor, eval_estack, eva
l_dstack) = autoencoder_model_3(patch_tensor)

print "Encoded:", encoded_tensor.get_shape().as_list()
print "Reconstructed:", reconstructed_tensor.get_shape().as_list()
```

```
Encoded: [None, 128]
Reconstructed: [None, 27, 27, 3]
```

In [17]:
```
# Restore the (final) model
saver = tf.train.Saver()
saver.restore(sess, "autoencoder_models/final_model_v1/v150/model.ckpt")
```

In [19]:
```
def get_training_op(reconstructed_tensor, patch_tensor, lr_tensor):
    difference = reconstructed_tensor - patch_tensor
    #print difference.get_shape().as_list()
    square_difference = tf.square(difference)
    mean_squared_error = tf.reduce_mean(square_difference)
    slim.losses.add_loss(mean_squared_error)
    total_loss = slim.losses.get_total_loss()
    optimizer = tf.train.AdamOptimizer(learning_rate=lr_tensor)#learning_
rate=0.01)
    #optimizer = tf.train.AdamOptimizer()
    train_op = slim.learning.create_train_op(total_loss, optimizer)
    return (train_op, mean_squared_error)
```

In [20]:
```
lr_tensor = tf.placeholder(dtype='float32', shape=[])
(train_op, loss) = get_training_op(reconstructed_tensor, patch_tensor, lr
_tensor)
```

```
def train_loop(sess, patch_tensor, reconstructed_tensor, eval_reconstruct
ed_tensor, train_patches, test_patches, train_op, loss_tensor, epochs, ba
tch_size, reset=True):
    if not os.path.exists("imgs/train"):
        os.makedirs("imgs/train")
    if not os.path.exists("imgs/test"):
        os.makedirs("imgs/test")
    tr_loss = []
    tst_loss = []
    N = train_patches.shape[0]
    if reset:
        sess.run(tf.initialize_all_variables())
    for e in xrange(epochs):
        for i in xrange(0, N, batch_size):
            [total_loss, loss] = sess.run([train_op, loss_tensor], feed_d
ict={
                    lr_tensor:0.001,
                    patch_tensor:train_patches[i:i+batch_size],
                })
            step = i / batch_size
            if step % 50 == 0:
                [test_loss] = sess.run([loss_tensor], feed_dict={
                    patch_tensor:test_patches[:100],
                })
                test_loss = 0
                print "Epoch %d, step %d, total loss %f, training loss
%f, test_loss %f" % (e, step, total_loss, loss, test_loss)
                tr_loss.append(loss)
                tst_loss.append(test_loss)
                #print "Starting print"
                imgs = sess.run(eval_reconstructed_tensor, feed_dict={
                    patch_tensor:train_patches[:16],
                })
                #plt.figure()
                for i in range(16):
                    plt.subplot(4,4,i+1)
                    plt.imshow(imgs[i])
                plt.savefig("imgs/train/%d_%d.png" % (e, step))
                #print "Done print training"
                imgs = sess.run(eval_reconstructed_tensor, feed_dict={
                    patch_tensor:test_patches[:16],
                })
                #plt.figure()
                for i in range(16):
                    plt.subplot(4,4,i+1)
                    plt.imshow(imgs[i])
                plt.savefig("imgs/test/%d_%d.png" % (e, step))
                #print "Done print testing"
        # End-of-epoch printing
        [test_loss] = sess.run([loss_tensor], feed_dict={
            patch_tensor:test_patches[:100],
        })
        test_loss = 0
        print "End of epoch %d, training loss %f, test_loss %f" % (e, los
s, test_loss)
        # Save the model
        saver = tf.train.Saver()
```

```
        os.makedirs("autoencoder_models/autosave/v%d" % e)
        save_path = saver.save(sess, "autoencoder_models/autosave/v%d/mod
el.ckpt" % e)
        print "Saved to:", save_path
    #print "Final training loss %f" % (loss)
    return (tr_loss, tst_loss)
```

In [ ]:
```
tr_loss, tst_loss = train_loop(
    sess, patch_tensor, reconstructed_tensor, eval_reconstructed_tensor,
train_dict['patches'], test_patches, train_op, loss, 500, 100, reset=Fals
e)
```

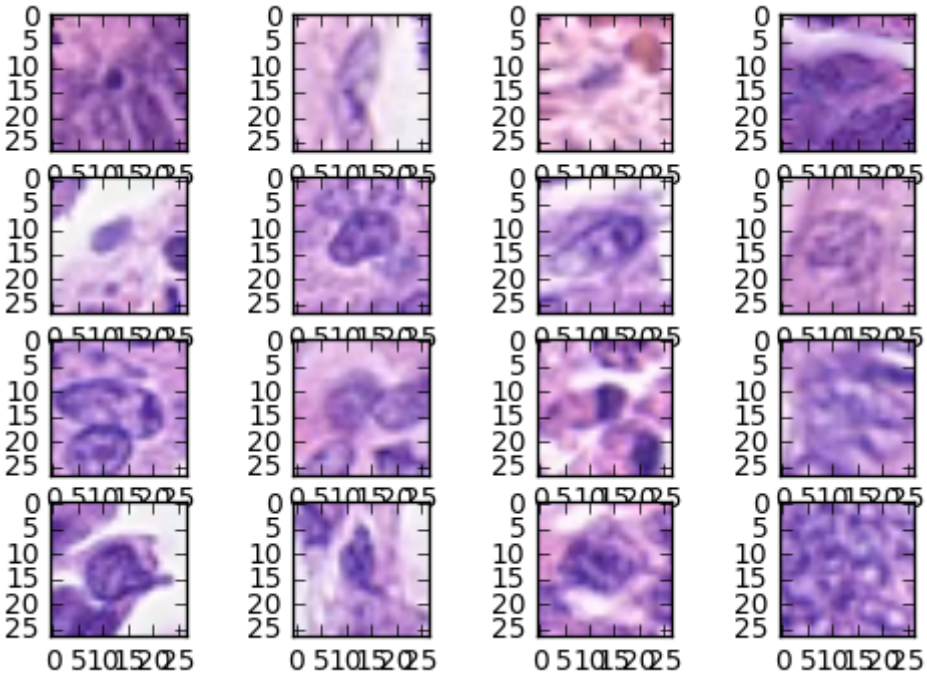
```
Epoch 35, step 50, total loss 0.004751, training loss 0.004751, test_loss
 0.000000
Epoch 35, step 100, total loss 0.004825, training loss 0.004825, test_los
s 0.000000
Epoch 35, step 150, total loss 0.004640, training loss 0.004640, test_los
s 0.000000
Epoch 35, step 200, total loss 0.004874, training loss 0.004874, test_los
s 0.000000
Epoch 35, step 250, total loss 0.004375, training loss 0.004375, test_los
s 0.000000
```

In [332]:
```
# Save the model
saver = tf.train.Saver()
save_path = saver.save(sess, "autoencoder_models/v3/model.ckpt")
print "Saved to:", save_path
```

```
Saved to: autoencoder_models/v3/model.ckpt
```

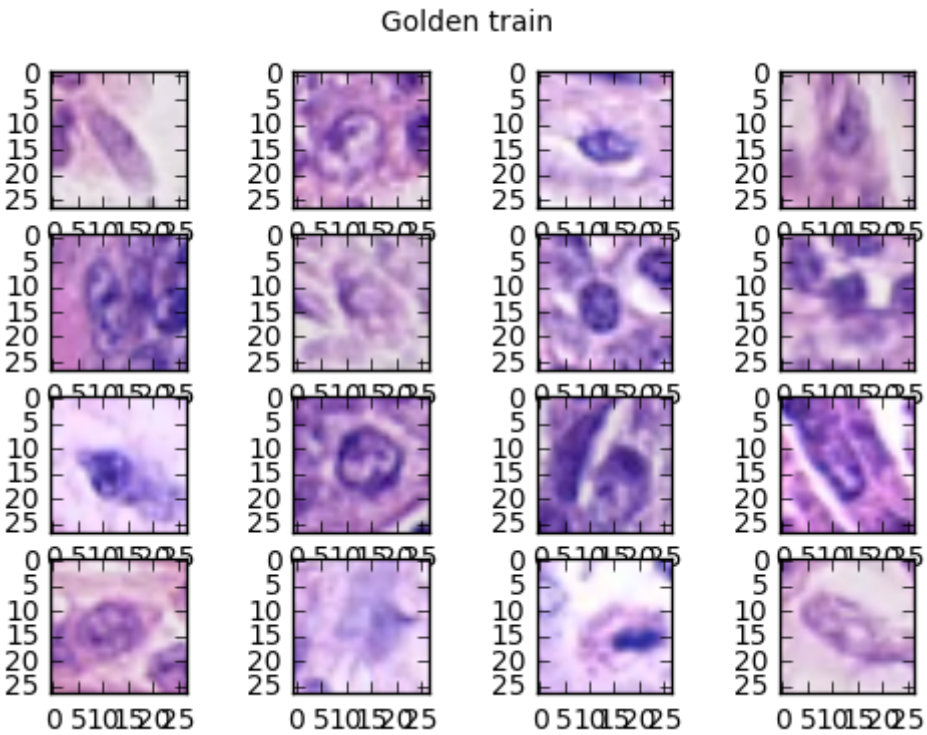
Compare Reconstructions to Originals

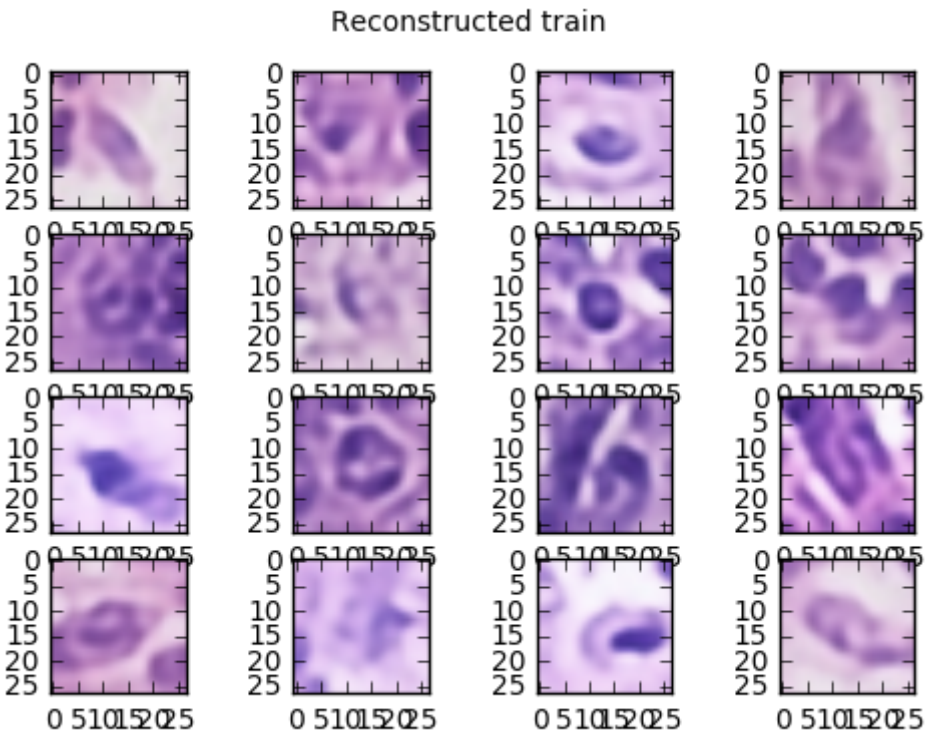
In [325]:
```
for i in range(16):
    plt.subplot(4,4,i+1)
    plt.imshow(train_dict['patches'][i])
plt.savefig("train_golden.png")
for i in range(16):
    plt.subplot(4,4,i+1)
    plt.imshow(test_patches[i])
plt.savefig("test_golden.png")
```

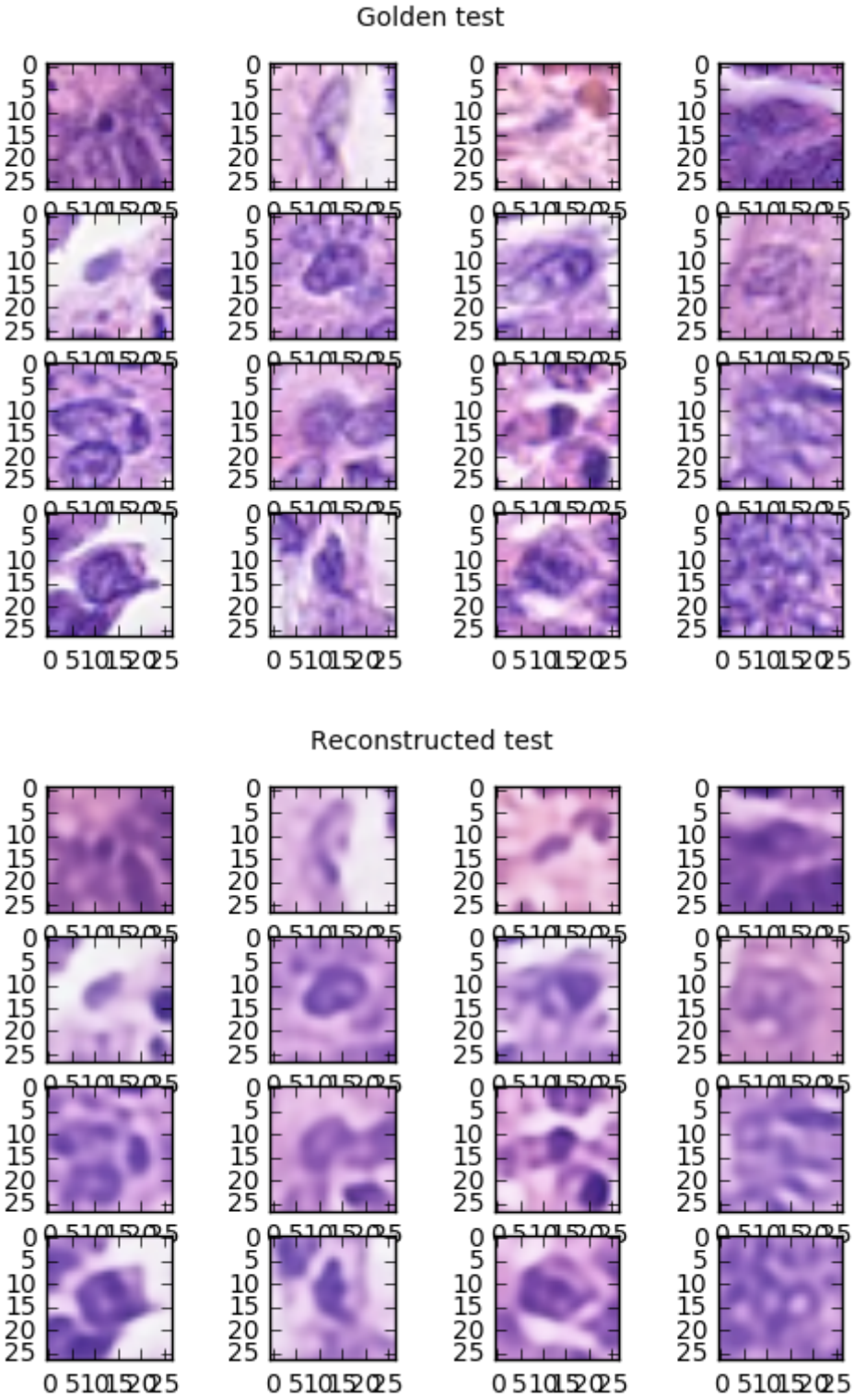


In [26]:
```
# Compare golden to best recreations
plt.figure()
for i in range(16):
    plt.subplot(4,4,i+1)
    plt.imshow(train_dict['patches'][i])
plt.suptitle("Golden train")

plt.figure()
imgs = sess.run(eval_reconstructed_tensor, feed_dict={
        patch_tensor:train_dict['patches'][:16],
    })
for i in range(16):
    plt.subplot(4,4,i+1)
    plt.imshow(imgs[i])
plt.suptitle("Reconstructed train")

plt.figure()
for i in range(16):
    plt.subplot(4,4,i+1)
    plt.imshow(test_patches[i])
plt.suptitle("Golden test")

plt.figure()
imgs = sess.run(eval_reconstructed_tensor, feed_dict={
        patch_tensor:test_patches[:16],
    })
for i in range(16):
    plt.subplot(4,4,i+1)
    plt.imshow(imgs[i])
plt.suptitle("Reconstructed test")
```

```
Out[26]: <matplotlib.text.Text at 0x7fd7f73e1b90>
```







Save Encodings

In [44]:
```
# Calculate the encodings for everything in the training and test sets
train_N = train_dict['patches'].shape[0]
test_N = test_patches.shape[0]
features_N = eval_encoded_tensor.get_shape().as_list()[1]
train_encodings = np.zeros((train_N, features_N))
test_encodings = np.zeros((test_N, features_N))
batch_size = 100 # We will do it in batches of 100 to save on memory

# First do the train set
for i in xrange(0, train_N, batch_size):
    train_encodings[i:i+batch_size] = sess.run(eval_encoded_tensor, feed_
dict={
            patch_tensor:train_dict['patches'][i:i+batch_size]
        })

# Next do the test set
for i in xrange(0, test_N, batch_size):
    test_encodings[i:i+batch_size] = sess.run(eval_encoded_tensor, feed_d
ict={
            patch_tensor:test_patches[i:i+batch_size]
        })
```

In [45]:
```
# Okay now we want to write these out to disk
np.save("train_autoencodings.npy", train_encodings)
np.save("test_autoencodings.npy", test_encodings)
```

Feature-Space Manipulations

In [19]:
```
categories
```

Out[19]:
```
['epithelial', 'fibroblast', 'inflammatory', 'others']
```