caffe (/github/BVLC/caffe/tree/master) / examples (/github/BVLC/caffe/tree/master/examples)

Fine-tuning a Pretrained Network for Style Recognition

In this example, we'll explore a common approach that is particularly useful in real-world applications: take a pre-trained Caffe network and fine-tune the parameters on your custom data.

The advantage of this approach is that, since pre-trained networks are learned on a large set of images, the intermediate layers capture the "semantics" of the general visual appearance. Think of it as a very powerful generic visual feature that you can treat as a black box. On top of that, only a relatively small amount of data is needed for good performance on the target task.

First, we will need to prepare the data. This involves the following parts: (1) Get the ImageNet ilsvrc pretrained model with the provided shell scripts. (2) Download a subset of the overall Flickr style dataset for this demo. (3) Compile the downloaded Flickr dataset into a database that Caffe can then consume.

In [1]:

```
caffe_root = '../'  # this file should be run from {caffe_root}/examples (otherwise change this line)

import sys
sys.path.insert(0, caffe_root + 'python')
import caffe

caffe.set_device(0)
caffe.set_mode_gpu()

import numpy as np
from pylab import *
%matplotlib inline
import tempfile

# Helper function for deprocessing preprocessed images, e.g., for display.
def deprocess_net_image(image):
    image = image.copy()              # don't modify destructively
    image = image[::-1]               # BGR -> RGB
    image = image.transpose(1, 2, 0)  # CHW -> HWC
    image += [123, 117, 104]          # (approximately) undo mean subtraction

    # clamp values in [0, 255]
    image[image < 0], image[image > 255] = 0, 255

    # round and cast from float32 to uint8
    image = np.round(image)
    image = np.require(image, dtype=np.uint8)

    return image
```

1. Setup and dataset download

Download data required for this exercise.

- `get_ilsvrc_aux.sh` to download the ImageNet data mean, labels, etc.
- `download_model_binary.py` to download the pretrained reference model
- `finetune_flickr_style/assemble_data.py` downloadsd the style training and testing data

We'll download just a small subset of the full dataset for this exercise: just 2000 of the 80K images, from 5 of the 20 style categories. (To download the full dataset, set `full_dataset = True` in the cell below.)

In [2]:

```
# Download just a small subset of the data for this exercise.
# (2000 of 80K images, 5 of 20 labels.)
# To download the entire dataset, set `full_dataset = True`.
full_dataset = False
if full_dataset:
    NUM_STYLE_IMAGES = NUM_STYLE_LABELS = -1
else:
    NUM_STYLE_IMAGES = 2000
    NUM_STYLE_LABELS = 5

# This downloads the ilsvrc auxiliary data (mean file, etc),
# and a subset of 2000 images for the style recognition task.
import os
os.chdir(caffe_root)  # run scripts from caffe root
!data/ilsvrc12/get_ilsvrc_aux.sh
!scripts/download_model_binary.py models/bvlc_reference_caffenet
!python examples/finetune_flickr_style/assemble_data.py \
    --workers=-1  --seed=1701 \
    --images=$NUM_STYLE_IMAGES  --label=$NUM_STYLE_LABELS
# back to examples
os.chdir('examples')
```

```
Downloading...
--2016-02-24 00:28:36--  http://dl.caffe.berkeleyvision.org/caffe_ilsvrc12.
tar.gz
Resolving dl.caffe.berkeleyvision.org (dl.caffe.berkeleyvision.org)... 169.
229.222.251
Connecting to dl.caffe.berkeleyvision.org (dl.caffe.berkeleyvision.org)|169
.229.222.251|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 17858008 (17M) [application/octet-stream]
Saving to: 'caffe_ilsvrc12.tar.gz'

100%[======================================>] 17,858,008   112MB/s   in 0.2
s

2016-02-24 00:28:36 (112 MB/s) - 'caffe_ilsvrc12.tar.gz' saved [17858008/
17858008]

Unzipping...
Done.
Model already exists.
Downloading 2000 images with 7 workers...
Writing train/val for 1996 successfully downloaded images.
```

Define `weights`, the path to the ImageNet pretrained weights we just downloaded, and make sure it exists.

In [3]:

```
import os
weights = caffe_root + 'models/bvlc_reference_caffenet/bvlc_reference_caffenet.caffemode
l'
assert os.path.exists(weights)
```

Load the 1000 ImageNet labels from `ilsvrc12/synset_words.txt`, and the 5 style labels from `finetune_flickr_style/style_names.txt`.

In [4]:

```
# Load ImageNet labels to imagenet_labels
imagenet_label_file = caffe_root + 'data/ilsvrc12/synset_words.txt'
imagenet_labels = list(np.loadtxt(imagenet_label_file, str, delimiter='\t'))
assert len(imagenet_labels) == 1000
print 'Loaded ImageNet labels:\n', '\n'.join(imagenet_labels[:10] + ['...'])

# Load style labels to style_labels
style_label_file = caffe_root + 'examples/finetune_flickr_style/style_names.txt'
style_labels = list(np.loadtxt(style_label_file, str, delimiter='\n'))
if NUM_STYLE_LABELS > 0:
    style_labels = style_labels[:NUM_STYLE_LABELS]
print '\nLoaded style labels:\n', ', '.join(style_labels)
```

```
Loaded ImageNet labels:
n01440764 tench, Tinca tinca
n01443537 goldfish, Carassius auratus
n01484850 great white shark, white shark, man-eater, man-eating shark, Carc
harodon carcharias
n01491361 tiger shark, Galeocerdo cuvieri
n01494475 hammerhead, hammerhead shark
n01496331 electric ray, crampfish, numbfish, torpedo
n01498041 stingray
n01514668 cock
n01514859 hen
n01518878 ostrich, Struthio camelus
...

Loaded style labels:
Detailed, Pastel, Melancholy, Noir, HDR
```

2. Defining and running the nets

We'll start by defining `caffenet`, a function which initializes the *CaffeNet* architecture (a minor variant on *AlexNet*), taking arguments specifying the data and number of output classes.

In [5]:

```
from caffe import layers as L
from caffe import params as P

weight_param = dict(lr_mult=1, decay_mult=1)
bias_param   = dict(lr_mult=2, decay_mult=0)
learned_param = [weight_param, bias_param]

frozen_param = [dict(lr_mult=0)] * 2

def conv_relu(bottom, ks, nout, stride=1, pad=0, group=1,
              param=learned_param,
              weight_filler=dict(type='gaussian', std=0.01),
              bias_filler=dict(type='constant', value=0.1)):
    conv = L.Convolution(bottom, kernel_size=ks, stride=stride,
                         num_output=nout, pad=pad, group=group,
                         param=param, weight_filler=weight_filler,
                         bias_filler=bias_filler)
    return conv, L.ReLU(conv, in_place=True)

def fc_relu(bottom, nout, param=learned_param,
            weight_filler=dict(type='gaussian', std=0.005),
            bias_filler=dict(type='constant', value=0.1)):
    fc = L.InnerProduct(bottom, num_output=nout, param=param,
                        weight_filler=weight_filler,
                        bias_filler=bias_filler)
    return fc, L.ReLU(fc, in_place=True)

def max_pool(bottom, ks, stride=1):
    return L.Pooling(bottom, pool=P.Pooling.MAX, kernel_size=ks, stride=stride)

def caffenet(data, label=None, train=True, num_classes=1000,
             classifier_name='fc8', learn_all=False):
    """Returns a NetSpec specifying CaffeNet, following the original proto text
       specification (./models/bvlc_reference_caffenet/train_val.prototxt)."""
    n = caffe.NetSpec()
    n.data = data
    param = learned_param if learn_all else frozen_param
    n.conv1, n.relu1 = conv_relu(n.data, 11, 96, stride=4, param=param)
    n.pool1 = max_pool(n.relu1, 3, stride=2)
    n.norm1 = L.LRN(n.pool1, local_size=5, alpha=1e-4, beta=0.75)
    n.conv2, n.relu2 = conv_relu(n.norm1, 5, 256, pad=2, group=2, param=param)
    n.pool2 = max_pool(n.relu2, 3, stride=2)
    n.norm2 = L.LRN(n.pool2, local_size=5, alpha=1e-4, beta=0.75)
    n.conv3, n.relu3 = conv_relu(n.norm2, 3, 384, pad=1, param=param)
    n.conv4, n.relu4 = conv_relu(n.relu3, 3, 384, pad=1, group=2, param=param)
    n.conv5, n.relu5 = conv_relu(n.relu4, 3, 256, pad=1, group=2, param=param)
    n.pool5 = max_pool(n.relu5, 3, stride=2)
    n.fc6, n.relu6 = fc_relu(n.pool5, 4096, param=param)
    if train:
        n.drop6 = fc7input = L.Dropout(n.relu6, in_place=True)
    else:
        fc7input = n.relu6
    n.fc7, n.relu7 = fc_relu(fc7input, 4096, param=param)
    if train:
        n.drop7 = fc8input = L.Dropout(n.relu7, in_place=True)
```

```
    else:
        fc8input = n.relu7
    # always learn fc8 (param=learned_param)
    fc8 = L.InnerProduct(fc8input, num_output=num_classes, param=learned_param)
    # give fc8 the name specified by argument `classifier_name`
    n.__setattr__(classifier_name, fc8)
    if not train:
        n.probs = L.Softmax(fc8)
    if label is not None:
        n.label = label
        n.loss = L.SoftmaxWithLoss(fc8, n.label)
        n.acc = L.Accuracy(fc8, n.label)
    # write the net to a temporary file and return its filename
    with tempfile.NamedTemporaryFile(delete=False) as f:
        f.write(str(n.to_proto()))
        return f.name
```

Now, let's create a *CaffeNet* that takes unlabeled "dummy data" as input, allowing us to set its input images externally and see what ImageNet classes it predicts.

In [6]:

```
dummy_data = L.DummyData(shape=dict(dim=[1, 3, 227, 227]))
imagenet_net_filename = caffenet(data=dummy_data, train=False)
imagenet_net = caffe.Net(imagenet_net_filename, weights, caffe.TEST)
```

Define a function `style_net` which calls `caffenet` on data from the Flickr style dataset.

The new network will also have the *CaffeNet* architecture, with differences in the input and output:

- the input is the Flickr style data we downloaded, provided by an `ImageData` layer
- the output is a distribution over 20 classes rather than the original 1000 ImageNet classes
- the classification layer is renamed from `fc8` to `fc8_flickr` to tell Caffe not to load the original classifier (`fc8`) weights from the ImageNet-pretrained model

In [7]:

```
def style_net(train=True, learn_all=False, subset=None):
    if subset is None:
        subset = 'train' if train else 'test'
    source = caffe_root + 'data/flickr_style/%s.txt' % subset
    transform_param = dict(mirror=train, crop_size=227,
        mean_file=caffe_root + 'data/ilsvrc12/imagenet_mean.binaryproto')
    style_data, style_label = L.ImageData(
        transform_param=transform_param, source=source,
        batch_size=50, new_height=256, new_width=256, ntop=2)
    return caffenet(data=style_data, label=style_label, train=train,
                    num_classes=NUM_STYLE_LABELS,
                    classifier_name='fc8_flickr',
                    learn_all=learn_all)
```

Use the `style_net` function defined above to initialize `untrained_style_net`, a *CaffeNet* with input images from the style dataset and weights from the pretrained ImageNet model.

Call `forward` on `untrained_style_net` to get a batch of style training data.

In [8]:

```
untrained_style_net = caffe.Net(style_net(train=False, subset='train'),
                                weights, caffe.TEST)
untrained_style_net.forward()
style_data_batch = untrained_style_net.blobs['data'].data.copy()
style_label_batch = np.array(untrained_style_net.blobs['label'].data, dtype=np.int32)
```

Pick one of the style net training images from the batch of 50 (we'll arbitrarily choose #8 here). Display it, then run it through `imagenet_net`, the ImageNet-pretrained network to view its top 5 predicted classes from the 1000 ImageNet classes.

Below we chose an image where the network's predictions happen to be reasonable, as the image is of a beach, and "sandbar" and "seashore" both happen to be ImageNet-1000 categories. For other images, the predictions won't be this good, sometimes due to the network actually failing to recognize the object(s) present in the image, but perhaps even more often due to the fact that not all images contain an object from the (somewhat arbitrarily chosen) 1000 ImageNet categories. Modify the `batch_index` variable by changing its default setting of 8 to another value from 0-49 (since the batch size is 50) to see predictions for other images in the batch. (To go beyond this batch of 50 images, first rerun the *above* cell to load a fresh batch of data into `style_net`.)

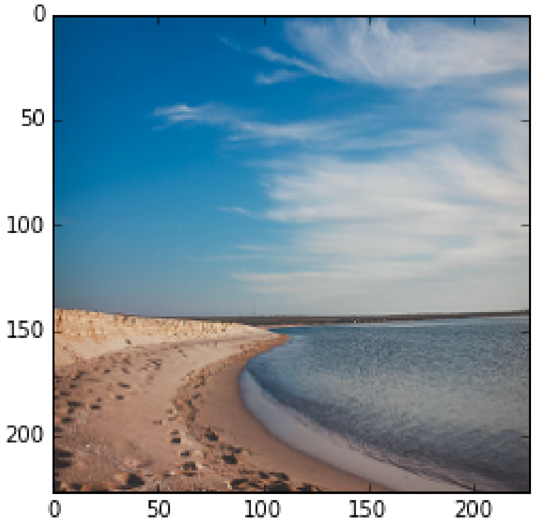
In [9]:

```
def disp_preds(net, image, labels, k=5, name='ImageNet'):
    input_blob = net.blobs['data']
    net.blobs['data'].data[0, ...] = image
    probs = net.forward(start='conv1')['probs'][0]
    top_k = (-probs).argsort()[:k]
    print 'top %d predicted %s labels =' % (k, name)
    print '\n'.join('\t(%d) %5.2f%% %s' % (i+1, 100*probs[p], labels[p])
                    for i, p in enumerate(top_k))

def disp_imagenet_preds(net, image):
    disp_preds(net, image, imagenet_labels, name='ImageNet')

def disp_style_preds(net, image):
    disp_preds(net, image, style_labels, name='style')
```

In [10]:

```
batch_index = 8
image = style_data_batch[batch_index]
plt.imshow(deprocess_net_image(image))
print 'actual label =', style_labels[style_label_batch[batch_index]]
```

```
actual label = Melancholy
```



In [11]:

```
disp_imagenet_preds(imagenet_net, image)
```

```
top 5 predicted ImageNet labels =
        (1) 69.89% n09421951 sandbar, sand bar
        (2) 21.76% n09428293 seashore, coast, seacoast, sea-coast
        (3)  3.22% n02894605 breakwater, groin, groyne, mole, bulwark, seaw
all, jetty
        (4)  1.89% n04592741 wing
        (5)  1.23% n09332890 lakeside, lakeshore
```

We can also look at `untrained_style_net`'s predictions, but we won't see anything interesting as its classifier hasn't been trained yet.

In fact, since we zero-initialized the classifier (see `caffenet` definition -- no `weight_filler` is passed to the final `InnerProduct` layer), the softmax inputs should be all zero and we should therefore see a predicted probability of 1/N for each label (for N labels). Since we set N = 5, we get a predicted probability of 20% for each class.

In [12]:

```
disp_style_preds(untrained_style_net, image)
```

```
top 5 predicted style labels =
        (1) 20.00% Detailed
        (2) 20.00% Pastel
        (3) 20.00% Melancholy
        (4) 20.00% Noir
        (5) 20.00% HDR
```

We can also verify that the activations in layer fc7 immediately before the classification layer are the same as (or very close to) those in the ImageNet-pretrained model, since both models are using the same pretrained weights in the conv1 through fc7 layers.

In [13]:

```
diff = untrained_style_net.blobs['fc7'].data[0] - imagenet_net.blobs['fc7'].data[0]
error = (diff ** 2).sum()
assert error < 1e-8
```

Delete untrained_style_net to save memory. (Hang on to imagenet_net as we'll use it again later.)

In [14]:

```
del untrained_style_net
```

3. Training the style classifier

Now, we'll define a function solver to create our Caffe solvers, which are used to train the network (learn its weights). In this function we'll set values for various parameters used for learning, display, and "snapshotting" -- see the inline comments for explanations of what they mean. You may want to play with some of the learning parameters to see if you can improve on the results here!

In [15]:

```
from caffe.proto import caffe_pb2

def solver(train_net_path, test_net_path=None, base_lr=0.001):
    s = caffe_pb2.SolverParameter()

    # Specify locations of the train and (maybe) test networks.
    s.train_net = train_net_path
    if test_net_path is not None:
        s.test_net.append(test_net_path)
        s.test_interval = 1000  # Test after every 1000 training iterations.
        s.test_iter.append(100) # Test on 100 batches each time we test.

    # The number of iterations over which to average the gradient.
    # Effectively boosts the training batch size by the given factor, without
    # affecting memory utilization.
    s.iter_size = 1

    s.max_iter = 100000     # # of times to update the net (training iterations)

    # Solve using the stochastic gradient descent (SGD) algorithm.
    # Other choices include 'Adam' and 'RMSProp'.
    s.type = 'SGD'

    # Set the initial learning rate for SGD.
    s.base_lr = base_lr

    # Set `lr_policy` to define how the learning rate changes during training.
    # Here, we 'step' the learning rate by multiplying it by a factor `gamma`
    # every `stepsize` iterations.
    s.lr_policy = 'step'
    s.gamma = 0.1
    s.stepsize = 20000

    # Set other SGD hyperparameters. Setting a non-zero `momentum` takes a
    # weighted average of the current gradient and previous gradients to make
    # learning more stable. L2 weight decay regularizes learning, to help prevent
    # the model from overfitting.
    s.momentum = 0.9
    s.weight_decay = 5e-4

    # Display the current training loss and accuracy every 1000 iterations.
    s.display = 1000

    # Snapshots are files used to store networks we've trained.  Here, we'll
    # snapshot every 10K iterations -- ten times during training.
    s.snapshot = 10000
    s.snapshot_prefix = caffe_root + 'models/finetune_flickr_style/finetune_flickr_style
'

    # Train on the GPU.  Using the CPU to train large networks is very slow.
    s.solver_mode = caffe_pb2.SolverParameter.GPU

    # Write the solver to a temporary file and return its filename.
    with tempfile.NamedTemporaryFile(delete=False) as f:
        f.write(str(s))
```

```
        return f.name
```

Now we'll invoke the solver to train the style net's classification layer.

For the record, if you want to train the network using only the command line tool, this is the command:

`build/tools/caffe train \ -solver models/finetune_flickr_style/solver.prototxt \ -weights models/bvlc_reference_caffenet/bvlc_reference_caffenet.caffemodel \ -gpu 0`

However, we will train using Python in this example.

We'll first define `run_solvers`, a function that takes a list of solvers and steps each one in a round robin manner, recording the accuracy and loss values each iteration. At the end, the learned weights are saved to a file.

In [16]:

```
def run_solvers(niter, solvers, disp_interval=10):
    """Run solvers for niter iterations,
       returning the loss and accuracy recorded each iteration.
       `solvers` is a list of (name, solver) tuples."""
    blobs = ('loss', 'acc')
    loss, acc = ({name: np.zeros(niter) for name, _ in solvers}
                 for _ in blobs)
    for it in range(niter):
        for name, s in solvers:
            s.step(1)  # run a single SGD step in Caffe
            loss[name][it], acc[name][it] = (s.net.blobs[b].data.copy()
                                             for b in blobs)
        if it % disp_interval == 0 or it + 1 == niter:
            loss_disp = '; '.join('%s: loss=%.3f, acc=%2d%%' %
                                  (n, loss[n][it], np.round(100*acc[n][it]))
                                  for n, _ in solvers)
            print '%3d) %s' % (it, loss_disp)
    # Save the learned weights from both nets.
    weight_dir = tempfile.mkdtemp()
    weights = {}
    for name, s in solvers:
        filename = 'weights.%s.caffemodel' % name
        weights[name] = os.path.join(weight_dir, filename)
        s.net.save(weights[name])
    return loss, acc, weights
```

Let's create and run solvers to train nets for the style recognition task. We'll create two solvers -- one (`style_solver`) will have its train net initialized to the ImageNet-pretrained weights (this is done by the call to the `copy_from` method), and the other (`scratch_style_solver`) will start from a *randomly* initialized net.

During training, we should see that the ImageNet pretrained net is learning faster and attaining better accuracies than the scratch net.

In [17]:

```
niter = 200  # number of iterations to train

# Reset style_solver as before.
style_solver_filename = solver(style_net(train=True))
style_solver = caffe.get_solver(style_solver_filename)
style_solver.net.copy_from(weights)

# For reference, we also create a solver that isn't initialized from
# the pretrained ImageNet weights.
scratch_style_solver_filename = solver(style_net(train=True))
scratch_style_solver = caffe.get_solver(scratch_style_solver_filename)

print 'Running solvers for %d iterations...' % niter
solvers = [('pretrained', style_solver),
           ('scratch', scratch_style_solver)]
loss, acc, weights = run_solvers(niter, solvers)
print 'Done.'

train_loss, scratch_train_loss = loss['pretrained'], loss['scratch']
train_acc, scratch_train_acc = acc['pretrained'], acc['scratch']
style_weights, scratch_style_weights = weights['pretrained'], weights['scratch']

# Delete solvers to save memory.
del style_solver, scratch_style_solver, solvers
```

```
Running solvers for 200 iterations...
  0) pretrained: loss=1.609, acc=28%; scratch: loss=1.609, acc=28%
 10) pretrained: loss=1.293, acc=52%; scratch: loss=1.626, acc=14%
 20) pretrained: loss=1.110, acc=56%; scratch: loss=1.646, acc=10%
 30) pretrained: loss=1.084, acc=60%; scratch: loss=1.616, acc=20%
 40) pretrained: loss=0.898, acc=64%; scratch: loss=1.588, acc=26%
 50) pretrained: loss=1.024, acc=54%; scratch: loss=1.607, acc=32%
 60) pretrained: loss=0.925, acc=66%; scratch: loss=1.616, acc=20%
 70) pretrained: loss=0.861, acc=74%; scratch: loss=1.598, acc=24%
 80) pretrained: loss=0.967, acc=60%; scratch: loss=1.588, acc=30%
 90) pretrained: loss=1.274, acc=52%; scratch: loss=1.608, acc=20%
100) pretrained: loss=1.113, acc=62%; scratch: loss=1.588, acc=30%
110) pretrained: loss=0.922, acc=62%; scratch: loss=1.578, acc=36%
120) pretrained: loss=0.918, acc=62%; scratch: loss=1.599, acc=20%
130) pretrained: loss=0.959, acc=58%; scratch: loss=1.594, acc=22%
140) pretrained: loss=1.228, acc=50%; scratch: loss=1.608, acc=14%
150) pretrained: loss=0.727, acc=76%; scratch: loss=1.623, acc=16%
160) pretrained: loss=1.074, acc=66%; scratch: loss=1.607, acc=20%
170) pretrained: loss=0.887, acc=60%; scratch: loss=1.614, acc=20%
180) pretrained: loss=0.961, acc=62%; scratch: loss=1.614, acc=18%
190) pretrained: loss=0.737, acc=76%; scratch: loss=1.613, acc=18%
199) pretrained: loss=0.836, acc=70%; scratch: loss=1.614, acc=16%
Done.
```

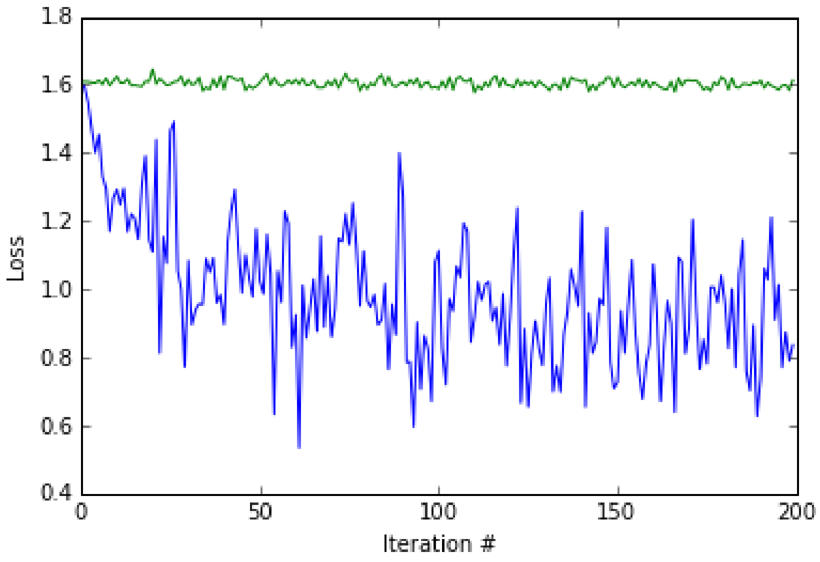
Let's look at the training loss and accuracy produced by the two training procedures. Notice how quickly the ImageNet pretrained model's loss value (blue) drops, and that the randomly initialized model's loss value (green) barely (if at all) improves from training only the classifier layer.

In [18]:

```
plot(np.vstack([train_loss, scratch_train_loss]).T)
xlabel('Iteration #')
ylabel('Loss')
```

Out[18]:

<matplotlib.text.Text at 0x7f75d49e1090>

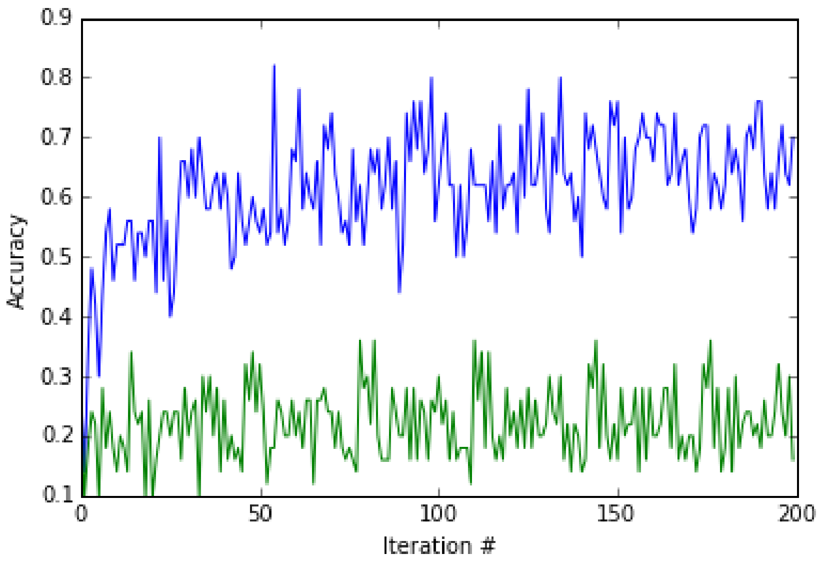


In [19]:

```
plot(np.vstack([train_acc, scratch_train_acc]).T)
xlabel('Iteration #')
ylabel('Accuracy')
```

Out[19]:

<matplotlib.text.Text at 0x7f75d49e1a90>



Let's take a look at the testing accuracy after running 200 iterations of training. Note that we're classifying among 5 classes, giving chance accuracy of 20%. We expect both results to be better than chance accuracy (20%), and we further expect the result from training using the ImageNet pretraining initialization to be much better than the one from training from scratch. Let's see.

In [20]:

```
def eval_style_net(weights, test_iters=10):
    test_net = caffe.Net(style_net(train=False), weights, caffe.TEST)
    accuracy = 0
    for it in xrange(test_iters):
        accuracy += test_net.forward()['acc']
    accuracy /= test_iters
    return test_net, accuracy
```

In [21]:

```
test_net, accuracy = eval_style_net(style_weights)
print 'Accuracy, trained from ImageNet initialization: %3.1f%%' % (100*accuracy, )
scratch_test_net, scratch_accuracy = eval_style_net(scratch_style_weights)
print 'Accuracy, trained from   random initialization: %3.1f%%' % (100*scratch_accurac
y, )
```

```
Accuracy, trained from ImageNet initialization: 50.0%
Accuracy, trained from   random initialization: 23.6%
```

4. End-to-end finetuning for style

Finally, we'll train both nets again, starting from the weights we just learned. The only difference this time is that we'll be learning the weights "end-to-end" by turning on learning in *all* layers of the network, starting from the RGB `conv1` filters directly applied to the input image. We pass the argument `learn_all=True` to the `style_net` function defined earlier in this notebook, which tells the function to apply a positive (non-zero) `lr_mult` value for all parameters. Under the default, `learn_all=False`, all parameters in the pretrained layers (`conv1` through `fc7`) are frozen (`lr_mult = 0`), and we learn only the classifier layer `fc8_flickr`.

Note that both networks start at roughly the accuracy achieved at the end of the previous training session, and improve significantly with end-to-end training. To be more scientific, we'd also want to follow the same additional training procedure *without* the end-to-end training, to ensure that our results aren't better simply because we trained for twice as long. Feel free to try this yourself!

In [22]:

```
end_to_end_net = style_net(train=True, learn_all=True)

# Set base_lr to 1e-3, the same as last time when learning only the classifier.
# You may want to play around with different values of this or other
# optimization parameters when fine-tuning.  For example, if learning diverges
# (e.g., the loss gets very large or goes to infinity/NaN), you should try
# decreasing base_lr (e.g., to 1e-4, then 1e-5, etc., until you find a value
# for which learning does not diverge).
base_lr = 0.001

style_solver_filename = solver(end_to_end_net, base_lr=base_lr)
style_solver = caffe.get_solver(style_solver_filename)
style_solver.net.copy_from(style_weights)

scratch_style_solver_filename = solver(end_to_end_net, base_lr=base_lr)
scratch_style_solver = caffe.get_solver(scratch_style_solver_filename)
scratch_style_solver.net.copy_from(scratch_style_weights)

print 'Running solvers for %d iterations...' % niter
solvers = [('pretrained, end-to-end', style_solver),
           ('scratch, end-to-end', scratch_style_solver)]
_, _, finetuned_weights = run_solvers(niter, solvers)
print 'Done.'

style_weights_ft = finetuned_weights['pretrained, end-to-end']
scratch_style_weights_ft = finetuned_weights['scratch, end-to-end']

# Delete solvers to save memory.
del style_solver, scratch_style_solver, solvers
```

```
Running solvers for 200 iterations...
  0) pretrained, end-to-end: loss=0.781, acc=64%; scratch, end-to-end: loss
=1.585, acc=28%
 10) pretrained, end-to-end: loss=1.178, acc=62%; scratch, end-to-end: loss
=1.638, acc=14%
 20) pretrained, end-to-end: loss=1.084, acc=60%; scratch, end-to-end: loss
=1.637, acc= 8%
 30) pretrained, end-to-end: loss=0.902, acc=76%; scratch, end-to-end: loss
=1.600, acc=20%
 40) pretrained, end-to-end: loss=0.865, acc=64%; scratch, end-to-end: loss
=1.574, acc=26%
 50) pretrained, end-to-end: loss=0.888, acc=60%; scratch, end-to-end: loss
=1.604, acc=26%
 60) pretrained, end-to-end: loss=0.538, acc=78%; scratch, end-to-end: loss
=1.555, acc=34%
 70) pretrained, end-to-end: loss=0.717, acc=72%; scratch, end-to-end: loss
=1.563, acc=30%
 80) pretrained, end-to-end: loss=0.695, acc=74%; scratch, end-to-end: loss
=1.502, acc=42%
 90) pretrained, end-to-end: loss=0.708, acc=68%; scratch, end-to-end: loss
=1.523, acc=26%
100) pretrained, end-to-end: loss=0.432, acc=78%; scratch, end-to-end: loss
=1.500, acc=38%
110) pretrained, end-to-end: loss=0.611, acc=78%; scratch, end-to-end: loss
=1.618, acc=18%
120) pretrained, end-to-end: loss=0.610, acc=76%; scratch, end-to-end: loss
```

```
120) pretrained, end-to-end: loss=0.610, acc=76%; scratch, end-to-end: loss
=1.473, acc=30%
130) pretrained, end-to-end: loss=0.471, acc=78%; scratch, end-to-end: loss
=1.488, acc=26%
140) pretrained, end-to-end: loss=0.500, acc=76%; scratch, end-to-end: loss
=1.514, acc=38%
150) pretrained, end-to-end: loss=0.476, acc=80%; scratch, end-to-end: loss
=1.452, acc=46%
160) pretrained, end-to-end: loss=0.368, acc=82%; scratch, end-to-end: loss
=1.419, acc=34%
170) pretrained, end-to-end: loss=0.556, acc=76%; scratch, end-to-end: loss
=1.583, acc=36%
180) pretrained, end-to-end: loss=0.574, acc=72%; scratch, end-to-end: loss
=1.556, acc=22%
190) pretrained, end-to-end: loss=0.360, acc=88%; scratch, end-to-end: loss
=1.429, acc=44%
199) pretrained, end-to-end: loss=0.458, acc=78%; scratch, end-to-end: loss
=1.370, acc=44%
Done.
```

Let's now test the end-to-end finetuned models. Since all layers have been optimized for the style recognition task at hand, we expect both nets to get better results than the ones above, which were achieved by nets with only their classifier layers trained for the style task (on top of either ImageNet pretrained or randomly initialized weights).

In [23]:

```
test_net, accuracy = eval_style_net(style_weights_ft)
print 'Accuracy, finetuned from ImageNet initialization: %3.1f%%' % (100*accuracy, )
scratch_test_net, scratch_accuracy = eval_style_net(scratch_style_weights_ft)
print 'Accuracy, finetuned from   random initialization: %3.1f%%' % (100*scratch_accur
acy, )
```

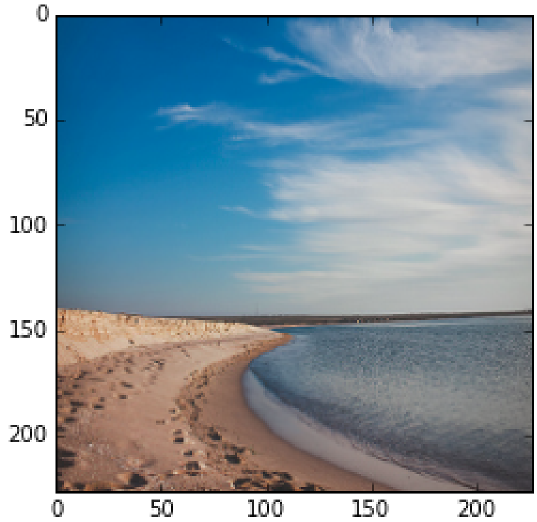
```
Accuracy, finetuned from ImageNet initialization: 53.6%
Accuracy, finetuned from   random initialization: 39.2%
```

We'll first look back at the image we started with and check our end-to-end trained model's predictions.

In [24]:

```
plt.imshow(deprocess_net_image(image))
disp_style_preds(test_net, image)
```

```
top 5 predicted style labels =
	(1) 55.67% Melancholy
	(2) 27.21% HDR
	(3) 16.46% Pastel
	(4)  0.63% Detailed
	(5)  0.03% Noir
```



Whew, that looks a lot better than before! But note that this image was from the training set, so the net got to see its label at training time.

Finally, we'll pick an image from the test set (an image the model hasn't seen) and look at our end-to-end finetuned style model's predictions for it.

In [25]:

```
batch_index = 1
image = test_net.blobs['data'].data[batch_index]
plt.imshow(deprocess_net_image(image))
print 'actual label =', style_labels[int(test_net.blobs['label'].data[batch_index])]
```

```
actual label = Pastel
```

In [26]:

```
disp_style_preds(test_net, image)
```

```
top 5 predicted style labels =
        (1) 99.76% Pastel
        (2)  0.13% HDR
        (3)  0.11% Detailed
        (4)  0.00% Melancholy
        (5)  0.00% Noir
```

We can also look at the predictions of the network trained from scratch. We see that in this case, the scratch network also predicts the correct label for the image (*Pastel*), but is much less confident in its prediction than the pretrained net.

In [27]:

```
disp_style_preds(scratch_test_net, image)
```

```
top 5 predicted style labels =
        (1) 49.81% Pastel
        (2) 19.76% Detailed
        (3) 17.06% Melancholy
        (4) 11.66% HDR
        (5)  1.72% Noir
```

Of course, we can again look at the ImageNet model's predictions for the above image:

In [28]:

```
disp_imagenet_preds(imagenet_net, image)
```

```
top 5 predicted ImageNet labels =
        (1) 34.90% n07579787 plate
        (2) 21.63% n04263257 soup bowl
        (3) 17.75% n07875152 potpie
        (4)  5.72% n07711569 mashed potato
        (5)  5.27% n07584110 consomme
```

So we did finetuning and it is awesome. Let's take a look at what kind of results we are able to get with a longer, more complete run of the style recognition dataset. Note: the below URL might be occassionally down because it is run on a research machine.

http://demo.vislab.berkeleyvision.org/ (http://demo.vislab.berkeleyvision.org/)