In [ ]:
```
#COMMENT IF NOT USING COLAB VM

# This mounts your Google Drive to the Colab VM.
#from google.colab import drive
#drive.mount('/content/drive')

# TODO: Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'DeepLearning/assignments/assignment3/'
#FOLDERNAME = None
#assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
#import sys
#sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# This downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
#%cd /content/drive/My\ Drive/$FOLDERNAME/cs6353/datasets/
#!bash get_datasets.sh
#%cd /content/drive/My\ Drive/$FOLDERNAME
```

In [ ]:
```
# #UNCOMMENT IF USING CADE
# import os
# ##### Request a GPU #####
# ## This function locates an available gpu for usage. In addition, this funct
ion reserves a specificed
# ## memory space exclusively for your account. The memory reservation prevent
s the decrement in computational
# ## speed when other users try to allocate memory on the same gpu in the shar
ed systems, i.e., CADE machines.
# ## Note: If you use your own system which has a GPU with less than 4GB of me
mory, remember to change the
# ## specified mimimum memory.
# def define_gpu_to_use(minimum_memory_mb = 3500):
#     thres_memory = 600 #
#     gpu_to_use = None
#     try:
#         os.environ['CUDA_VISIBLE_DEVICES']
#         print('GPU already assigned before: ' + str(os.environ['CUDA_VISIBLE
_DEVICES']))
#         return
#     except:
#         pass

#     for i in range(16):
#         free_memory = !nvidia-smi --query-gpu=memory.free -i $i --format=cs
v,nounits,noheader
#         if free_memory[0] == 'No devices were found':
#             break
#         free_memory = int(free_memory[0])

#         if free_memory>minimum_memory_mb-thres_memory:
#             gpu_to_use = i
#             break

#     if gpu_to_use is None:
#         print('Could not find any GPU available with the required free memor
y of ' + str(minimum_memory_mb) \
#               + 'MB. Please use a different system for this assignment.')
#     else:
#         os.environ['CUDA_VISIBLE_DEVICES'] = str(gpu_to_use)
#         print('Chosen GPU: ' + str(gpu_to_use))

# ## Request a gpu and reserve the memory space
# define_gpu_to_use(4000)
```

Image Captioning with RNNs

In this exercise you will implement a vanilla recurrent neural networks and use them it to train a model that can generate novel captions for images.

In [7]:
```
# As usual, a bit of setup
import time, os, json
import numpy as np
import matplotlib.pyplot as plt

from cs6353.gradient_check import eval_numerical_gradient, eval_numerical_grad
ient_array
from cs6353.rnn_layers import *
from cs6353.captioning_solver import CaptioningSolver
from cs6353.classifiers.rnn import CaptioningRNN
from cs6353.coco_utils import load_coco_data, sample_coco_minibatch, decode_ca
ptions
from cs6353.image_utils import image_from_url

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipyt
hon
%load_ext autoreload
%autoreload 2

def rel_error(x, y):
    """ returns relative error """
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

Install h5py

The COCO dataset we will be using is stored in HDF5 format. To load HDF5 files, we will need to install the `h5py` Python package. Check if h5py is already installed:

In [3]:
```
import h5py
```

If the modual is not found, you will need to install it now. From the command line, run:
`pip install h5py`
If you receive a permissions error, you may need to run the command as root:
`sudo pip install h5py`

You can also run commands directly from the Jupyter notebook by prefixing the command with the "!" character:

In [4]:
```
!pip install h5py
```

```
Requirement already satisfied: h5py in c:\users\justi\anaconda3\envs\cs6353\l
ib\site-packages (3.1.0)
Requirement already satisfied: numpy>=1.12 in c:\users\justi\anaconda3\envs\c
s6353\lib\site-packages (from h5py) (1.19.5)
Requirement already satisfied: cached-property in c:\users\justi\anaconda3\en
vs\cs6353\lib\site-packages (from h5py) (1.5.2)
```

Microsoft COCO

For this exercise we will use the 2014 release of the Microsoft COCO dataset (http://mscoco.org/) which has become the standard testbed for image captioning. The dataset consists of 80,000 training images and 40,000 validation images, each annotated with 5 captions written by workers on Amazon Mechanical Turk.

You should have already downloaded the data by changing to the `cs6353/datasets` directory and running the script `get_assignment3_data.sh`. If you haven't yet done so, run that script now. Warning: the COCO data download is ~1GB.

We have preprocessed the data and extracted features for you already. For all images we have extracted features from the fc7 layer of the VGG-16 network pretrained on ImageNet; these features are stored in the files `train2014_vgg16_fc7.h5` and `val2014_vgg16_fc7.h5` respectively. To cut down on processing time and memory requirements, we have reduced the dimensionality of the features from 4096 to 512; these features can be found in the files `train2014_vgg16_fc7_pca.h5` and `val2014_vgg16_fc7_pca.h5`.

The raw images take up a lot of space (nearly 20GB) so we have not included them in the download. However all images are taken from Flickr, and URLs of the training and validation images are stored in the files `train2014_urls.txt` and `val2014_urls.txt` respectively. This allows you to download images on the fly for visualization. Since images are downloaded on-the-fly, **you must be connected to the internet to view images**.

Dealing with strings is inefficient, so we will work with an encoded version of the captions. Each word is assigned an integer ID, allowing us to represent a caption by a sequence of integers. The mapping between integer IDs and words is in the file `coco2014_vocab.json`, and you can use the function `decode_captions` from the file `cs6353/coco_utils.py` to convert numpy arrays of integer IDs back into strings.

There are a couple special tokens that we add to the vocabulary. We prepend a special `<START>` token and append an `<END>` token to the beginning and end of each caption respectively. Rare words are replaced with a special `<UNK>` token (for "unknown"). In addition, since we want to train with minibatches containing captions of different lengths, we pad short captions with a special `<NULL>` token after the `<END>` token and don't compute loss or gradient for `<NULL>` tokens. Since they are a bit of a pain, we have taken care of all implementation details around special tokens for you.

You can load all of the MS-COCO data (captions, features, URLs, and vocabulary) using the `load_coco_data` function from the file `cs6353/coco_utils.py`. Run the following cell to do so:

In [8]:
```
# Load COCO data from disk; this returns a dictionary
# We'll work with dimensionality-reduced features for this notebook, but feel
# free to experiment with the original features by changing the flag below.
data = load_coco_data(pca_features=True)

# Print out all the keys and values from the data dictionary
for k, v in data.items():
    if type(v) == np.ndarray:
        print(k, type(v), v.shape, v.dtype)
    else:
        print(k, type(v), len(v))
```

```
train_captions <class 'numpy.ndarray'> (400135, 17) int32
train_image_idxs <class 'numpy.ndarray'> (400135,) int32
val_captions <class 'numpy.ndarray'> (195954, 17) int32
val_image_idxs <class 'numpy.ndarray'> (195954,) int32
train_features <class 'numpy.ndarray'> (82783, 512) float32
val_features <class 'numpy.ndarray'> (40504, 512) float32
idx_to_word <class 'list'> 1004
word_to_idx <class 'dict'> 1004
train_urls <class 'numpy.ndarray'> (82783,) <U63
val_urls <class 'numpy.ndarray'> (40504,) <U63
```

Look at the data

It is always a good idea to look at examples from the dataset before working with it.

You can use the `sample_coco_minibatch` function from the file `cs6353/coco_utils.py` to sample minibatches of data from the data structure returned from `load_coco_data`. Run the following to sample a small minibatch of training data and show the images and their captions. Running it multiple times and looking at the results helps you to get a sense of the dataset.

Note that we decode the captions using the `decode_captions` function and that we download the images on-the-fly using their Flickr URL, so **you must be connected to the internet to view images**.

In [33]:
```
# Sample a minibatch and show the images and captions
batch_size = 3

captions, features, urls = sample_coco_minibatch(data, batch_size=batch_size)
for i, (caption, url) in enumerate(zip(captions, urls)):
    plt.imshow(image_from_url(url))
    plt.axis('off')
    caption_str = decode_captions(caption, data['idx_to_word'])
    plt.title(caption_str)
    plt.show()
```

<START> a desk and chair with a computer and a lamp <END>

<START> a <UNK> bike leaning up against the side of a building <END>

<START> bicycles parked at a <UNK> of motor boats <END>

Recurrent Neural Networks

As discussed in lecture, we will use recurrent neural network (RNN) language models for image captioning. The file `cs6353/rnn_layers.py` contains implementations of different layer types that are needed for recurrent neural networks, and the file `cs6353/classifiers/rnn.py` uses these layers to implement an image captioning model.

We will first implement different types of RNN layers in `cs6353/rnn_layers.py` .

Vanilla RNN: step forward

Open the file `cs6353/rnn_layers.py` . This file implements the forward and backward passes for different types of layers that are commonly used in recurrent neural networks.

First implement the function `rnn_step_forward` which implements the forward pass for a single timestep of a vanilla recurrent neural network. After doing so run the following to check your implementation. You should see errors on the order of e-8 or less.

In [13]:
```
N, D, H = 3, 10, 4

x = np.linspace(-0.4, 0.7, num=N*D).reshape(N, D)
prev_h = np.linspace(-0.2, 0.5, num=N*H).reshape(N, H)
Wx = np.linspace(-0.1, 0.9, num=D*H).reshape(D, H)
Wh = np.linspace(-0.3, 0.7, num=H*H).reshape(H, H)
b = np.linspace(-0.2, 0.4, num=H)

next_h, _ = rnn_step_forward(x, prev_h, Wx, Wh, b)
expected_next_h = np.asarray([
  [-0.58172089, -0.50182032, -0.41232771, -0.31410098],
  [ 0.66854692,  0.79562378,  0.87755553,  0.92795967],
  [ 0.97934501,  0.99144213,  0.99646691,  0.99854353]])

print('next_h error: ', rel_error(expected_next_h, next_h))
```

```
next_h error:  6.292421426471037e-09
```

Vanilla RNN: step backward

In the file `cs6353/rnn_layers.py` implement the `rnn_step_backward` function. After doing so run the following to numerically gradient check your implementation. You should see errors on the order of `e-8` or less.

In [16]:
```
from cs6353.rnn_layers import rnn_step_forward, rnn_step_backward
np.random.seed(231)
N, D, H = 4, 5, 6
x = np.random.randn(N, D)
h = np.random.randn(N, H)
Wx = np.random.randn(D, H)
Wh = np.random.randn(H, H)
b = np.random.randn(H)

out, cache = rnn_step_forward(x, h, Wx, Wh, b)

dnext_h = np.random.randn(*out.shape)

fx = lambda x: rnn_step_forward(x, h, Wx, Wh, b)[0]
fh = lambda prev_h: rnn_step_forward(x, h, Wx, Wh, b)[0]
fWx = lambda Wx: rnn_step_forward(x, h, Wx, Wh, b)[0]
fWh = lambda Wh: rnn_step_forward(x, h, Wx, Wh, b)[0]
fb = lambda b: rnn_step_forward(x, h, Wx, Wh, b)[0]

dx_num = eval_numerical_gradient_array(fx, x, dnext_h)
dprev_h_num = eval_numerical_gradient_array(fh, h, dnext_h)
dWx_num = eval_numerical_gradient_array(fWx, Wx, dnext_h)
dWh_num = eval_numerical_gradient_array(fWh, Wh, dnext_h)
db_num = eval_numerical_gradient_array(fb, b, dnext_h)

dx, dprev_h, dWx, dWh, db = rnn_step_backward(dnext_h, cache)

print('dx error: ', rel_error(dx_num, dx))
print('dprev_h error: ', rel_error(dprev_h_num, dprev_h))
print('dWx error: ', rel_error(dWx_num, dWx))
print('dWh error: ', rel_error(dWh_num, dWh))
print('db error: ', rel_error(db_num, db))
```

```
dx error:  3.004984354606141e-10
dprev_h error:  2.633205333189269e-10
dWx error:  9.684083573724284e-10
dWh error:  3.355162782632426e-10
db error:  1.5956895526227225e-11
```

Vanilla RNN: forward

Now that you have implemented the forward and backward passes for a single timestep of a vanilla RNN, you will combine these pieces to implement a RNN that processes an entire sequence of data.

In the file `cs6353/rnn_layers.py`, implement the function `rnn_forward`. This should be implemented using the `rnn_step_forward` function that you defined above. After doing so run the following to check your implementation. You should see errors on the order of `e-7` or less.

In [17]:
```
N, T, D, H = 2, 3, 4, 5

x = np.linspace(-0.1, 0.3, num=N*T*D).reshape(N, T, D)
h0 = np.linspace(-0.3, 0.1, num=N*H).reshape(N, H)
Wx = np.linspace(-0.2, 0.4, num=D*H).reshape(D, H)
Wh = np.linspace(-0.4, 0.1, num=H*H).reshape(H, H)
b = np.linspace(-0.7, 0.1, num=H)

h, _ = rnn_forward(x, h0, Wx, Wh, b)
expected_h = np.asarray([
  [
    [-0.42070749, -0.27279261, -0.11074945,  0.05740409,  0.22236251],
    [-0.39525808, -0.22554661, -0.0409454,   0.14649412,  0.32397316],
    [-0.42305111, -0.24223728, -0.04287027,  0.15997045,  0.35014525],
  ],
  [
    [-0.55857474, -0.39065825, -0.19198182,  0.02378408,  0.23735671],
    [-0.27150199, -0.07088804,  0.13562939,  0.33099728,  0.50158768],
    [-0.51014825, -0.30524429, -0.06755202,  0.17806392,  0.40333043]]])
print('h error: ', rel_error(expected_h, h))
```

```
h error:  7.728466180186066e-08
```

Vanilla RNN: backward

In the file `cs6353/rnn_layers.py`, implement the backward pass for a vanilla RNN in the function `rnn_backward`. This should run back-propagation over the entire sequence, making calls to the `rnn_step_backward` function that you defined earlier. You should see errors on the order of e-6 or less.

In [19]:
```
np.random.seed(231)

N, D, T, H = 2, 5, 10, 5

x = np.random.randn(N, T, D)
h0 = np.random.randn(N, H)
Wx = np.random.randn(D, H)
Wh = np.random.randn(H, H)
b = np.random.randn(H)

out, cache = rnn_forward(x, h0, Wx, Wh, b)

dout = np.random.randn(*out.shape)

dx, dh0, dWx, dWh, db = rnn_backward(dout, cache)

fx = lambda x: rnn_forward(x, h0, Wx, Wh, b)[0]
fh0 = lambda h0: rnn_forward(x, h0, Wx, Wh, b)[0]
fWx = lambda Wx: rnn_forward(x, h0, Wx, Wh, b)[0]
fWh = lambda Wh: rnn_forward(x, h0, Wx, Wh, b)[0]
fb = lambda b: rnn_forward(x, h0, Wx, Wh, b)[0]

dx_num = eval_numerical_gradient_array(fx, x, dout)
dh0_num = eval_numerical_gradient_array(fh0, h0, dout)
dWx_num = eval_numerical_gradient_array(fWx, Wx, dout)
dWh_num = eval_numerical_gradient_array(fWh, Wh, dout)
db_num = eval_numerical_gradient_array(fb, b, dout)

print('dx error: ', rel_error(dx_num, dx))
print('dh0 error: ', rel_error(dh0_num, dh0))
print('dWx error: ', rel_error(dWx_num, dWx))
print('dWh error: ', rel_error(dWh_num, dWh))
print('db error: ', rel_error(db_num, db))
```

```
dx error:  4.551033782255309e-09
dh0 error:  8.95471327896557e-11
dWx error:  4.934224297058265e-10
dWh error:  4.6032473986829285e-09
db error:  6.207770928983651e-09
```

Word embedding: forward

In deep learning systems, we commonly represent words using vectors. Each word of the vocabulary will be associated with a vector, and these vectors will be learned jointly with the rest of the system.

In the file `cs6353/rnn_layers.py`, implement the function `word_embedding_forward` to convert words (represented by integers) into vectors. Run the following to check your implementation. You should see an error on the order of `e-8` or less.

In [20]:
```
N, T, V, D = 2, 4, 5, 3

x = np.asarray([[0, 3, 1, 2], [2, 1, 0, 3]])
W = np.linspace(0, 1, num=V*D).reshape(V, D)

out, _ = word_embedding_forward(x, W)
expected_out = np.asarray([
 [[ 0.,          0.07142857,  0.14285714],
  [ 0.64285714,  0.71428571,  0.78571429],
  [ 0.21428571,  0.28571429,  0.35714286],
  [ 0.42857143,  0.5,         0.57142857]],
 [[ 0.42857143,  0.5,         0.57142857],
  [ 0.21428571,  0.28571429,  0.35714286],
  [ 0.,          0.07142857,  0.14285714],
  [ 0.64285714,  0.71428571,  0.78571429]]])

print('out error: ', rel_error(expected_out, out))
```

```
out error:  1.0000000094736443e-08
```

Word embedding: backward

Implement the backward pass for the word embedding function in the function `word_embedding_backward`. After doing so run the following to numerically gradient check your implementation. You should see an error on the order of `e-11` or less.

In [21]:
```
np.random.seed(231)

N, T, V, D = 50, 3, 5, 6
x = np.random.randint(V, size=(N, T))
W = np.random.randn(V, D)

out, cache = word_embedding_forward(x, W)
dout = np.random.randn(*out.shape)
dW = word_embedding_backward(dout, cache)

f = lambda W: word_embedding_forward(x, W)[0]
dW_num = eval_numerical_gradient_array(f, W, dout)

print('dW error: ', rel_error(dW, dW_num))
```

```
dW error:  3.2774595693100364e-12
```

Temporal Affine layer

At every timestep we use an affine function to transform the RNN hidden vector at that timestep into scores for each word in the vocabulary. Because this is very similar to the affine layer that you implemented in assignment 2, we have provided this function for you in the `temporal_affine_forward` and `temporal_affine_backward` functions in the file `cs6353/rnn_layers.py`. Run the following to perform numeric gradient checking on the implementation. You should see errors on the order of e-9 or less.

In [22]:
```
np.random.seed(231)

# Gradient check for temporal affine layer
N, T, D, M = 2, 3, 4, 5
x = np.random.randn(N, T, D)
w = np.random.randn(D, M)
b = np.random.randn(M)

out, cache = temporal_affine_forward(x, w, b)

dout = np.random.randn(*out.shape)

fx = lambda x: temporal_affine_forward(x, w, b)[0]
fw = lambda w: temporal_affine_forward(x, w, b)[0]
fb = lambda b: temporal_affine_forward(x, w, b)[0]

dx_num = eval_numerical_gradient_array(fx, x, dout)
dw_num = eval_numerical_gradient_array(fw, w, dout)
db_num = eval_numerical_gradient_array(fb, b, dout)

dx, dw, db = temporal_affine_backward(dout, cache)

print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
dx error:  2.9215854231394017e-10
dw error:  1.5772169135951167e-10
db error:  3.252200556967514e-11
```

Temporal Softmax loss

In an RNN language model, at every timestep we produce a score for each word in the vocabulary. We know the ground-truth word at each timestep, so we use a softmax loss function to compute loss and gradient at each timestep. We sum the losses over time and average them over the minibatch.

However there is one wrinkle: since we operate over minibatches and different captions may have different lengths, we append `<NULL>` tokens to the end of each caption so they all have the same length. We don't want these `<NULL>` tokens to count toward the loss or gradient, so in addition to scores and ground-truth labels our loss function also accepts a `mask` array that tells it which elements of the scores count towards the loss.

Since this is very similar to the softmax loss function you implemented in assignment 1, we have implemented this loss function for you; look at the `temporal_softmax_loss` function in the file `cs6353/rnn_layers.py` .

Run the following cell to sanity check the loss and perform numeric gradient checking on the function. You should see an error for dx on the order of e-7 or less.

In [23]:
```
# Sanity check for temporal softmax loss
from cs6353.rnn_layers import temporal_softmax_loss

N, T, V = 100, 1, 10

def check_loss(N, T, V, p):
    x = 0.001 * np.random.randn(N, T, V)
    y = np.random.randint(V, size=(N, T))
    mask = np.random.rand(N, T) <= p
    print(temporal_softmax_loss(x, y, mask)[0])

check_loss(100, 1, 10, 1.0)   # Should be about 2.3
check_loss(100, 10, 10, 1.0)  # Should be about 23
check_loss(5000, 10, 10, 0.1) # Should be about 2.3

# Gradient check for temporal softmax loss
N, T, V = 7, 8, 9

x = np.random.randn(N, T, V)
y = np.random.randint(V, size=(N, T))
mask = (np.random.rand(N, T) > 0.5)

loss, dx = temporal_softmax_loss(x, y, mask, verbose=False)

dx_num = eval_numerical_gradient(lambda x: temporal_softmax_loss(x, y, mask)
[0], x, verbose=False)

print('dx error: ', rel_error(dx, dx_num))
```

```
2.3027781774290146
23.025985953127226
2.2643611790293394
dx error:  2.583585303524283e-08
```

RNN for image captioning

Now that you have implemented the necessary layers, you can combine them to build an image captioning model. Open the file `cs6353/classifiers/rnn.py` and look at the `CaptioningRNN` class.

Implement the forward and backward pass of the model in the `loss` function. For now you only need to implement the case where `cell_type='rnn'` for vanialla RNNs; you will implement the LSTM case later. After doing so, run the following to check your forward pass using a small test case; you should see error on the order of `e-10` or less.

In [26]:
```
N, D, W, H = 10, 20, 40, 40
word_to_idx = {'<NULL>': 0, 'cat': 2, 'dog': 3}
V = len(word_to_idx)
T = 13

model = CaptioningRNN(word_to_idx,
          input_dim=D,
          wordvec_dim=W,
          hidden_dim=H,
          cell_type='rnn',
          dtype=np.float64)

# Set all model parameters to fixed values
for k, v in model.params.items():
    model.params[k] = np.linspace(-1.4, 1.3, num=v.size).reshape(*v.shape)

features = np.linspace(-1.5, 0.3, num=(N * D)).reshape(N, D)
captions = (np.arange(N * T) % V).reshape(N, T)

loss, grads = model.loss(features, captions)
expected_loss = 9.83235591003

print('loss: ', loss)
print('expected loss: ', expected_loss)
print('difference: ', abs(loss - expected_loss))
```

```
loss:  9.809174730925447
expected loss:  9.83235591003
difference:  0.02318117910455264
```

Run the following cell to perform numeric gradient checking on the `CaptioningRNN` class; you should see errors around the order of `e-6` or less.

In [27]:
```
np.random.seed(231)

batch_size = 2
timesteps = 3
input_dim = 4
wordvec_dim = 6
hidden_dim = 6
word_to_idx = {'<NULL>': 0, 'cat': 2, 'dog': 3}
vocab_size = len(word_to_idx)

captions = np.random.randint(vocab_size, size=(batch_size, timesteps))
features = np.random.randn(batch_size, input_dim)

model = CaptioningRNN(word_to_idx,
          input_dim=input_dim,
          wordvec_dim=wordvec_dim,
          hidden_dim=hidden_dim,
          cell_type='rnn',
          dtype=np.float64,
        )

loss, grads = model.loss(features, captions)

for param_name in sorted(grads):
    f = lambda _: model.loss(features, captions)[0]
    param_grad_num = eval_numerical_gradient(f, model.params[param_name], verb
ose=False, h=1e-6)
    e = rel_error(param_grad_num, grads[param_name])
    print('%s relative error: %e' % (param_name, e))
```

```
W_embed relative error: 1.410848e-09
W_proj relative error: 4.900688e-09
W_vocab relative error: 1.879471e-09
Wh relative error: 6.212484e-09
Wx relative error: 6.125181e-07
b relative error: 6.015951e-10
b_proj relative error: 4.936157e-09
b_vocab relative error: 3.619792e-10
```

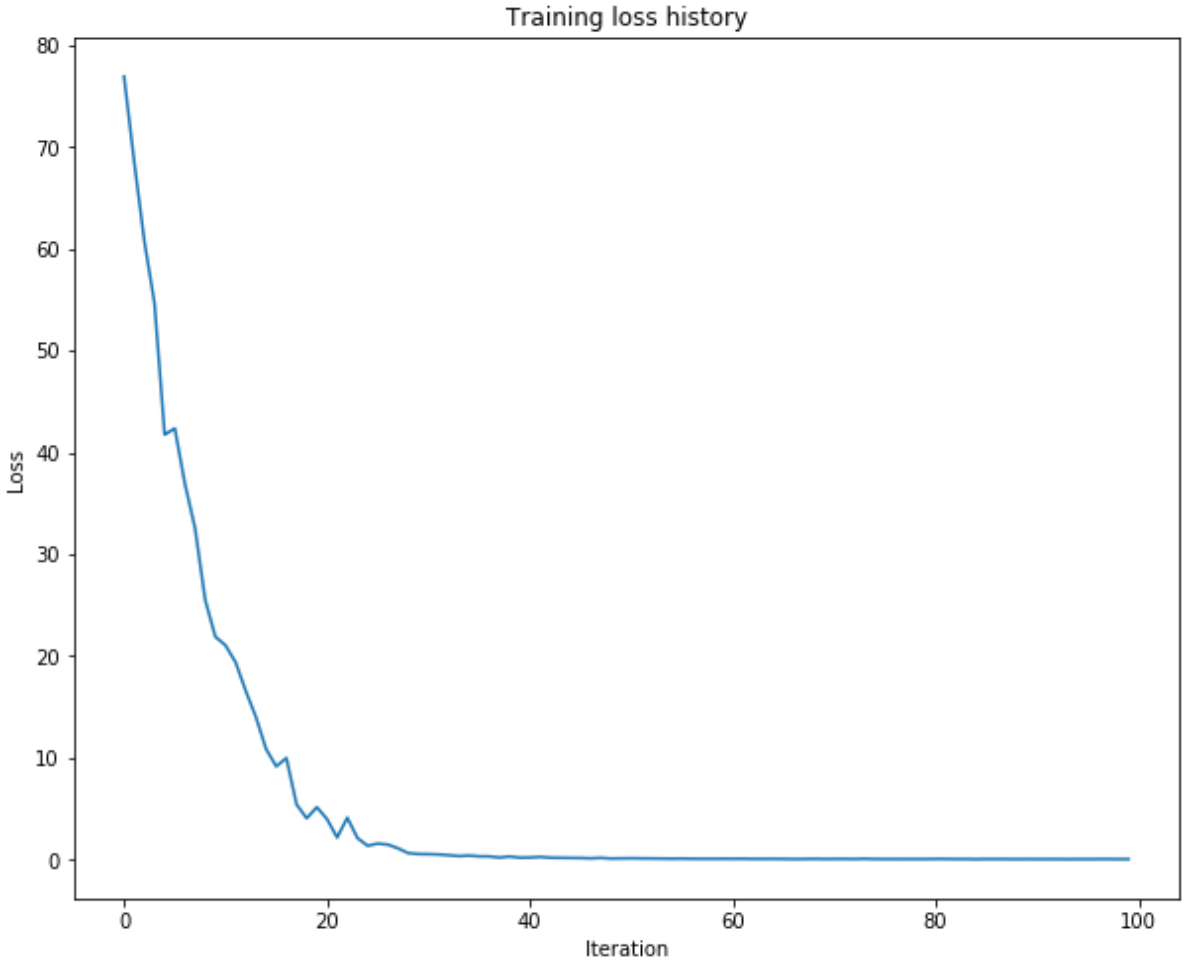
Overfit small data

Similar to the `Solver` class that we used to train image classification models on the previous assignment, on this assignment we use a `CaptioningSolver` class to train image captioning models. Open the file `cs6353/captioning_solver.py` and read through the `CaptioningSolver` class; it should look very familiar.

Once you have familiarized yourself with the API, run the following to make sure your model overfits a small sample of 100 training examples. You should see a final loss of less than 0.1.

In [31]:
```
np.random.seed(231)

small_data = load_coco_data(max_train=50)

small_rnn_model = CaptioningRNN(
          cell_type='rnn',
          word_to_idx=data['word_to_idx'],
          input_dim=data['train_features'].shape[1],
          hidden_dim=512,
          wordvec_dim=256,
        )

small_rnn_solver = CaptioningSolver(small_rnn_model, small_data,
           update_rule='adam',
           num_epochs=50,
           batch_size=25,
           optim_config={
             'learning_rate': 5e-3,
           },
           lr_decay=0.95,
           verbose=True, print_every=10,
         )

small_rnn_solver.train()

# Plot the training losses
plt.plot(small_rnn_solver.loss_history)
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.title('Training loss history')
plt.show()
```

```
(Iteration 1 / 100) loss: 76.913487
(Iteration 11 / 100) loss: 21.063153
(Iteration 21 / 100) loss: 4.016218
(Iteration 31 / 100) loss: 0.567106
(Iteration 41 / 100) loss: 0.239433
(Iteration 51 / 100) loss: 0.162020
(Iteration 61 / 100) loss: 0.111540
(Iteration 71 / 100) loss: 0.097585
(Iteration 81 / 100) loss: 0.099094
(Iteration 91 / 100) loss: 0.073977
```



Test-time sampling

Unlike classification models, image captioning models behave very differently at training time and at test time. At training time, we have access to the ground-truth caption, so we feed ground-truth words as input to the RNN at each timestep. At test time, we sample from the distribution over the vocabulary at each timestep, and feed the sample as input to the RNN at the next timestep.

In the file `cs6353/classifiers/rnn.py`, implement the `sample` method for test-time sampling. After doing so, run the following to sample from your overfitted model on both training and validation data. The samples on training data should be very good; the samples on validation data probably won't make sense.

In [32]:
```
for split in ['train', 'val']:
    minibatch = sample_coco_minibatch(small_data, split=split, batch_size=2)
    gt_captions, features, urls = minibatch
    gt_captions = decode_captions(gt_captions, data['idx_to_word'])

    sample_captions = small_rnn_model.sample(features)
    sample_captions = decode_captions(sample_captions, data['idx_to_word'])

    for gt_caption, sample_caption, url in zip(gt_captions, sample_captions, u
rls):
        plt.imshow(image_from_url(url))
        plt.title('%s\n%s\nGT:%s' % (split, sample_caption, gt_caption))
        plt.axis('off')
        plt.show()
```

train
a boy sitting with <UNK> on with a donut in his hand <END>
GT:<START> a boy sitting with <UNK> on with a donut in his hand <END>

train
a man <UNK> with a bright colorful kite <END>
GT:<START> a man <UNK> with a bright colorful kite <END>

val
two of <UNK> woman of a while in sun <UNK> <END>
GT:<START> a red and white light tower on a hill near the ocean <END>

val
to tracks with out of a <END>
GT:<START> a table filled with many assorted food items <END>

INLINE QUESTION 1

In our current image captioning setup, our RNN language model produces a word at every timestep as its output. However, an alternate way to pose the problem is to train the network to operate over *characters* (e.g. 'a', 'b', etc.) as opposed to words, so that at it every timestep, it receives the previous character as input and tries to predict the next character in the sequence. For example, the network might generate a caption like

'A', ' ', 'c', 'a', 't', ' ', 'o', 'n', ' ', 'a', ' ', 'b', 'e', 'd'

Can you describe one advantage of an image-captioning model that uses a character-level RNN? Can you also describe one disadvantage? HINT: there are several valid answers, but it might be useful to compare the parameter space of word-level and character-level models.

Answer

1. Pro. Char-level RNNs require less memory. The memory will only be 26 assuming it is using the English alphabet to build the words whereas if you were using a word-level RNN it would have to store every word in the dataset.
2. Con. Char-level RNNs have higher computational cost because they require bigger hidden layers and larger parameter-space. Usually a character level RNN will have more tokens than a word level model which means that it has to take into account dependencies between more tokens over more time steps. This means that they will need more hidden layers and parameters in order to successfully model dependencies.

In [ ]:
```
#COMMENT IF NOT USING COLAB VM

# This mounts your Google Drive to the Colab VM.
#from google.colab import drive
#drive.mount('/content/drive')

# TODO: Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'DeepLearning/assignments/assignment3/'
#FOLDERNAME = None
#assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
#import sys
#sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# This downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
#%cd /content/drive/My\ Drive/$FOLDERNAME/cs6353/datasets/
#!bash get_datasets.sh
#%cd /content/drive/My\ Drive/$FOLDERNAME
```

In [ ]:
```
# #UNCOMMENT IF USING CADE
# import os
# ##### Request a GPU #####
# ## This function locates an available gpu for usage. In addition, this funct
ion reserves a specificed
# ## memory space exclusively for your account. The memory reservation prevent
s the decrement in computational
# ## speed when other users try to allocate memory on the same gpu in the shar
ed systems, i.e., CADE machines.
# ## Note: If you use your own system which has a GPU with less than 4GB of me
mory, remember to change the
# ## specified mimimum memory.
# def define_gpu_to_use(minimum_memory_mb = 3500):
#     thres_memory = 600 #
#     gpu_to_use = None
#     try:
#         os.environ['CUDA_VISIBLE_DEVICES']
#         print('GPU already assigned before: ' + str(os.environ['CUDA_VISIBLE
_DEVICES']))
#         return
#     except:
#         pass

#     for i in range(16):
#         free_memory = !nvidia-smi --query-gpu=memory.free -i $i --format=cs
v,nounits,noheader
#         if free_memory[0] == 'No devices were found':
#             break
#         free_memory = int(free_memory[0])

#         if free_memory>minimum_memory_mb-thres_memory:
#             gpu_to_use = i
#             break

#     if gpu_to_use is None:
#         print('Could not find any GPU available with the required free memor
y of ' + str(minimum_memory_mb) \
#               + 'MB. Please use a different system for this assignment.')
#     else:
#         os.environ['CUDA_VISIBLE_DEVICES'] = str(gpu_to_use)
#         print('Chosen GPU: ' + str(gpu_to_use))

# ## Request a gpu and reserve the memory space
# define_gpu_to_use(4000)
```

Network Visualization (TensorFlow)

In this notebook we will explore the use of *image gradients* for generating new images.

When training a model, we define a loss function which measures our current unhappiness with the model's performance; we then use backpropagation to compute the gradient of the loss with respect to the model parameters, and perform gradient descent on the model parameters to minimize the loss.

Here we will do something slightly different. We will start from a convolutional neural network model which has been pretrained to perform image classification on the ImageNet dataset. We will use this model to define a loss function which quantifies our current unhappiness with our image, then use backpropagation to compute the gradient of this loss with respect to the pixels of the image. We will then keep the model fixed, and perform gradient descent *on the image* to synthesize a new image which minimizes the loss.

In this notebook we will explore three techniques for image generation:

1. **Saliency Maps**: Saliency maps are a quick way to tell which part of the image influenced the classification decision made by the network.
2. **Fooling Images**: We can perturb an input image so that it appears the same to humans, but will be misclassified by the pretrained network.
3. **Class Visualization**: We can synthesize an image to maximize the classification score of a particular class; this can give us some sense of what the network is looking for when it classifies images of that class.

This notebook uses **TensorFlow**; we have provided another notebook which explores the same concepts in PyTorch. You only need to complete one of these two notebooks.

In [1]:
```
# As usual, a bit of setup
import time, os, json
import numpy as np
import matplotlib.pyplot as plt
import tensorflow.compat.v1 as tf
from tensorflow.python.framework import ops

from cs6353.classifiers.squeezenet import SqueezeNet
from cs6353.data_utils import load_tiny_imagenet
from cs6353.image_utils import preprocess_image, deprocess_image
from cs6353.image_utils import SQUEEZENET_MEAN, SQUEEZENET_STD

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'
tf.disable_v2_behavior()

def get_session():
    """Create a session that dynamically allocates memory."""
    # See: https://www.tensorflow.org/tutorials/using_gpu#allowing_gpu_memory_
growth
    config = tf.ConfigProto()
    config.gpu_options.allow_growth = True
    session = tf.Session(config=config)
    return session

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipyt
hon
%load_ext autoreload
%autoreload 2
```

```
WARNING:tensorflow:From C:\Users\justi\anaconda3\envs\cs6353\lib\site-package
s\tensorflow\python\compat\v2_compat.py:101: disable_resource_variables (from
tensorflow.python.ops.variable_scope) is deprecated and will be removed in a
future version.
Instructions for updating:
non-resource variables are not supported in the long term
```

Pretrained Model

For all of our image generation experiments, we will start with a convolutional neural network which was pretrained to perform image classification on ImageNet. We can use any model here, but for the purposes of this assignment we will use SqueezeNet [1], which achieves accuracies comparable to AlexNet but with a significantly reduced parameter count and computational complexity.

Using SqueezeNet rather than AlexNet or VGG or ResNet means that we can easily perform all image generation experiments on CPU.

We have ported the PyTorch SqueezeNet model to TensorFlow; see: `cs6353/classifiers/squeezenet.py` for the model architecture.

To use SqueezeNet, you will need to first **download the weights** by descending into the `cs6353/datasets` directory and running `get_squeezenet_tf.sh` . Note that if you ran `get_assignment3_data.sh` then SqueezeNet will already be downloaded.

Once you've downloaded the Squeezenet model, we can load it into a new TensorFlow session:

[1] Iandola et al, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and < 0.5MB model size", arXiv 2016

In [2]:
```
tf.reset_default_graph()
sess = get_session()

SAVE_PATH = 'cs6353/datasets/squeezenet.ckpt'
if not os.path.exists(SAVE_PATH + ".index"):
    raise ValueError("You need to download SqueezeNet!")
model = SqueezeNet(save_path=SAVE_PATH, sess=sess)
```

```
INFO:tensorflow:Restoring parameters from cs6353/datasets/squeezenet.ckpt
```

Load some ImageNet images

We have provided a few example images from the validation set of the ImageNet ILSVRC 2012 Classification dataset. To download these images, descend into `cs6353/datasets/` and run `get_imagenet_val.sh` .

Since they come from the validation set, our pretrained model did not see these images during training.

Run the following cell to visualize some of these images, along with their ground-truth labels.

In [3]:
```
from cs6353.data_utils import load_imagenet_val
X_raw, y, class_names = load_imagenet_val(num=5)

plt.figure(figsize=(12, 6))
for i in range(5):
    plt.subplot(1, 5, i + 1)
    plt.imshow(X_raw[i])
    plt.title(class_names[y[i]])
    plt.axis('off')
plt.gcf().tight_layout()
```

Preprocess images

The input to the pretrained model is expected to be normalized, so we first preprocess the images by subtracting the pixelwise mean and dividing by the pixelwise standard deviation.

In [4]:
```
X = np.array([preprocess_image(img) for img in X_raw])
```

Saliency Maps

Using this pretrained model, we will compute class saliency maps as described in Section 3.1 of [2].

A **saliency map** tells us the degree to which each pixel in the image affects the classification score for that image. To compute it, we compute the gradient of the unnormalized score corresponding to the correct class (which is a scalar) with respect to the pixels of the image. If the image has shape `(H, W, 3)` then this gradient will also have shape `(H, W, 3)`; for each pixel in the image, this gradient tells us the amount by which the classification score will change if the pixel changes by a small amount. To compute the saliency map, we take the absolute value of this gradient, then take the maximum value over the 3 input channels; the final saliency map thus has shape `(H, W)` and all entries are nonnegative.

You will need to use the `model.scores` Tensor containing the scores for each input, and will need to feed in values for the `model.image` and `model.labels` placeholder when evaluating the gradient. Open the file `cs6353/classifiers/squeezenet.py` and read the documentation to make sure you understand how to use the model. For example usage, you can see the `loss` attribute.

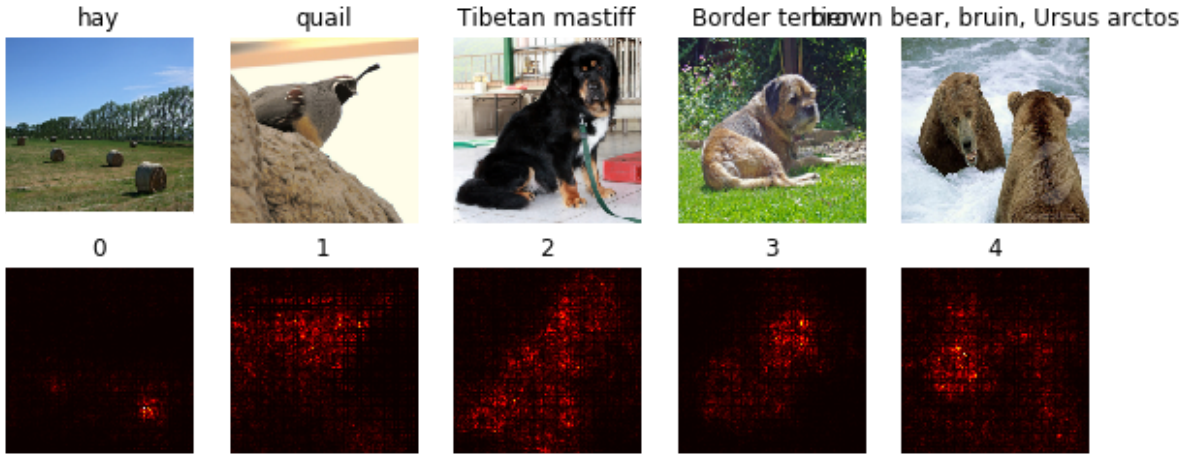
[2] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps", ICLR Workshop 2014.

In [17]:
```
def compute_saliency_maps(X, y, model):
    """
    Compute a class saliency map using the model for images X and labels y.

    Input:
    - X: Input images, numpy array of shape (N, H, W, 3)
    - y: Labels for X, numpy of shape (N,)
    - model: A SqueezeNet model that will be used to compute the saliency map.

    Returns:
    - saliency: A numpy array of shape (N, H, W) giving the saliency maps for
the
    input images.
    """
    saliency = None
    # Compute the score of the correct class for each example.
    # This gives a Tensor with shape [N], the number of examples.
    #
    # Note: this is equivalent to scores[np.arange(N), y] we used in NumPy
    # for computing vectorized losses.
    correct_scores = tf.gather_nd(model.scores,
                                  tf.stack((tf.range(X.shape[0]), model.label
s), axis=1))
    ##############################################################################
#####
    # TODO: Produce the saliency maps over a batch of images.
#
    #
#
    # 1) Compute the "loss" using the correct scores tensor provided for you.
#
    #    (We'll combine losses across a batch by summing)
#
    # 2) Use tf.gradients to compute the gradient of the loss with respect
#
    #    to the image (accessible via model.image).
#
    # 3) Compute the actual value of the gradient by a call to sess.run().
#
    #    You will need to feed in values for the placeholders model.image and
#
    #    model.labels.
#
    # 4) Finally, process the returned gradient to compute the saliency map.
#
    ##############################################################################
#####
    loss = correct_scores
    dimage = tf.gradients(loss, model.image)
    dimage_values = sess.run(dimage, feed_dict={model.image:X, model.labels:
y})[0]
    saliency = np.abs(dimage_values).max(axis=3)
    pass
    ##############################################################################
####
    #                               END OF YOUR CODE
```

```
    #
    ##############################################################################
####
    return saliency
```

Once you have completed the implementation in the cell above, run the following to visualize some class saliency maps on our example images from the ImageNet validation set:

In [18]:
```
def show_saliency_maps(X, y, mask):
    mask = np.asarray(mask)
    Xm = X[mask]
    ym = y[mask]

    saliency = compute_saliency_maps(Xm, ym, model)

    for i in range(mask.size):
        plt.subplot(2, mask.size, i + 1)
        plt.imshow(deprocess_image(Xm[i]))
        plt.axis('off')
        plt.title(class_names[ym[i]])
        plt.subplot(2, mask.size, mask.size + i + 1)
        plt.title(mask[i])
        plt.imshow(saliency[i], cmap=plt.cm.hot)
        plt.axis('off')
        plt.gcf().set_size_inches(10, 4)
    plt.show()

mask = np.arange(5)
show_saliency_maps(X, y, mask)
```



INLINE QUESTION

A friend of yours suggests that in order to find an image that maximizes the correct score, we can perform gradient ascent on the input image, but instead of the gradient we can actually use the saliency map in each step to update the image. Is this assertion true? Why or why not?

Answer: It is true because the saliency map is the gradient of the correct score with regards to the image which is the same as the gradient used to perform gradient ascent. Since the saliency map has the shape (H, W) it will work on grayscale iamges, but for images with more than one channel it might have a different behaviour because the saliency map is the max gradient across the channels.

Fooling Images

We can also use image gradients to generate "fooling images" as discussed in [3]. Given an image and a target class, we can perform gradient **ascent** over the image to maximize the target class, stopping when the network classifies the image as the target class. Implement the following function to generate fooling images.

[3] Szegedy et al, "Intriguing properties of neural networks", ICLR 2014

In [19]:
```
def make_fooling_image(X, target_y, model):
    """
    Generate a fooling image that is close to X, but that the model classifies
    as target_y.

    Inputs:
    - X: Input image, a numpy array of shape (1, 224, 224, 3)
    - target_y: An integer in the range [0, 1000)
    - model: Pretrained SqueezeNet model

    Returns:
    - X_fooling: An image that is close to X, but that is classifed as target_
y
    by the model.
    """

    # Make a copy of the input that we will modify
    X_fooling = X.copy()

    # Step size for the update
    learning_rate = 1

    ################################################################################
####
    # TODO: Generate a fooling image X_fooling that the model will classify as
#
    # the class target_y. Use gradient *ascent* on the target class score, usi
ng #
    # the model.scores Tensor to get the class scores for the model.image.   #
    # When computing an update step, first normalize the gradient:
#
    #   dX = learning_rate * g / ||g||_2
#
    #
#
    # You should write a training loop, where in each iteration, you make an
#
    # update to the input image X_fooling (don't modify X). The loop should
#
    # stop when the predicted class for the input is the same as target_y.
#
    #
#
    # HINT: It's good practice to define your TensorFlow graph operations
#
    # outside the loop, and then just make sess.run() calls in each iteration.
#
    #
#
    # HINT 2: For most examples, you should be able to generate a fooling imag
e  #
    # in fewer than 100 iterations of gradient ascent. You can print your
#
    # progress over iterations to check your algorithm.
#
    ################################################################################
```

```
####
    loss = model.scores[0][target_y]
    dfooling = tf.gradients(loss, model.image)
    image_result = model.image + learning_rate * dfooling/tf.norm(dfooling, or
d=2)
    max_score_label = tf.argmax(model.scores[0])

    for i in range(100):
        _image_result, _loss, _max_score_label = sess.run([image_result, loss,
max_score_label],
                                                           feed_dict={model.ima
ge: X_fooling})
        if _max_score_label == target_y:
            break
        X_fooling = _image_result[0]
    pass
    ################################################################################
####
    #                               END OF YOUR CODE
#
    ################################################################################
####
    return X_fooling
```

Run the following to generate a fooling image. You should ideally see at first glance no major difference between the original and fooling images, and the network should now make an incorrect prediction on the fooling one. However you should see a bit of random noise if you look at the 10x magnified difference between the original and fooling images. Feel free to change the `idx` variable to explore other images.

In [21]:
```
idx = 0
Xi = X[idx][None]
target_y = 6
X_fooling = make_fooling_image(Xi, target_y, model)

# Make sure that X_fooling is classified as y_target
scores = sess.run(model.scores, {model.image: X_fooling})
assert scores[0].argmax() == target_y, 'The network is not fooled!'

# Show original image, fooling image, and difference
orig_img = deprocess_image(Xi[0])
fool_img = deprocess_image(X_fooling[0])
# Rescale
plt.subplot(1, 4, 1)
plt.imshow(orig_img)
plt.axis('off')
plt.title(class_names[y[idx]])
plt.subplot(1, 4, 2)
plt.imshow(fool_img)
plt.title(class_names[target_y])
plt.axis('off')
plt.subplot(1, 4, 3)
plt.title('Difference')
plt.imshow(deprocess_image((Xi-X_fooling)[0]))
plt.axis('off')
plt.subplot(1, 4, 4)
plt.title('Magnified difference (10x)')
plt.imshow(deprocess_image(10 * (Xi-X_fooling)[0]))
plt.axis('off')
plt.gcf().tight_layout()
```

Class visualization

By starting with a random noise image and performing gradient ascent on a target class, we can generate an image that the network will recognize as the target class. This idea was first presented in [2]; [3] extended this idea by suggesting several regularization techniques that can improve the quality of the generated image.

Concretely, let I be an image and let y be a target class. Let $s_y(I)$ be the score that a convolutional network assigns to the image I for class y; note that these are raw unnormalized scores, not class probabilities. We wish to generate an image I^* that achieves a high score for the class y by solving the problem

$$I^* = \arg\max_I (s_y(I) - R(I))$$

where R is a (possibly implicit) regularizer (note the sign of $R(I)$ in the argmax: we want to minimize this regularization term). We can solve this optimization problem using gradient ascent, computing gradients with respect to the generated image. We will use (explicit) L2 regularization of the form

$$R(I) = \lambda \|I\|_2^2$$

and implicit regularization as suggested by [3] by periodically blurring the generated image. We can solve this problem using gradient ascent on the generated image.

In the cell below, complete the implementation of the `create_class_visualization` function.

[2] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps", ICLR Workshop 2014.

[3] Yosinski et al, "Understanding Neural Networks Through Deep Visualization", ICML 2015 Deep Learning Workshop

In [22]:
```
from scipy.ndimage.filters import gaussian_filter1d
def blur_image(X, sigma=1):
    X = gaussian_filter1d(X, sigma, axis=1)
    X = gaussian_filter1d(X, sigma, axis=2)
    return X
```

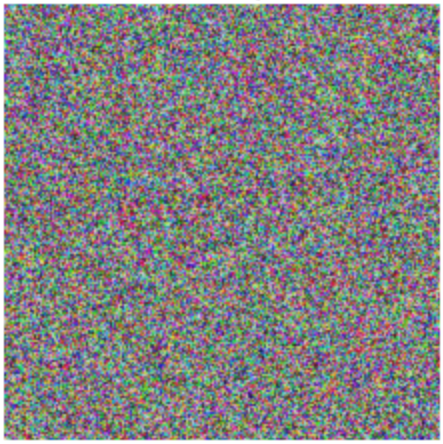
In [23]:
```
def create_class_visualization(target_y, model, **kwargs):
    """
    Generate an image to maximize the score of target_y under a pretrained mod
el.

    Inputs:
    - target_y: Integer in the range [0, 1000) giving the index of the class
    - model: A pretrained CNN that will be used to generate the image

    Keyword arguments:
    - l2_reg: Strength of L2 regularization on the image
    - learning_rate: How big of a step to take
    - num_iterations: How many iterations to use
    - blur_every: How often to blur the image as an implicit regularizer
    - max_jitter: How much to gjitter the image as an implicit regularizer
    - show_every: How often to show the intermediate result
    """
    l2_reg = kwargs.pop('l2_reg', 1e-3)
    learning_rate = kwargs.pop('learning_rate', 25)
    num_iterations = kwargs.pop('num_iterations', 100)
    blur_every = kwargs.pop('blur_every', 10)
    max_jitter = kwargs.pop('max_jitter', 16)
    show_every = kwargs.pop('show_every', 25)

    # We use a single image of random noise as a starting point
    X = 255 * np.random.rand(224, 224, 3)
    X = preprocess_image(X)[None]

    ################################################################################
    # TODO: Compute the loss and the gradient of the loss with respect to  #
    # the input image, model.image. We compute these outside the loop so   #
    # that we don't have to recompute the gradient graph at each iteration #
    #                                                                      #
    # Note: loss and grad should be TensorFlow Tensors, not numpy arrays!  #
    #                                                                      #
    # The loss is the score for the target label, target_y. You should     #
    # use model.scores to get the scores, and tf.gradients to compute  #
    # gradients. Don't forget the (subtracted) L2 regularization term!     #
    ################################################################################
    loss = model.scores[0][target_y] - l2_reg * tf.nn.l2_loss(model.image)
    grad = tf.gradients(loss, model.image)
    pass
    ################################################################################
##
    #                             END OF YOUR CODE
#
    ################################################################################
##


    for t in range(num_iterations):
        # Randomly jitter the image a bit; this gives slightly nicer results
        ox, oy = np.random.randint(-max_jitter, max_jitter+1, 2)
        X = np.roll(np.roll(X, ox, 1), oy, 2)

        ########################################################################
```

```
##
        # TODO: Use sess to compute the value of the gradient of the score for
#
        # class target_y with respect to the pixels of the image, and make a
#
        # gradient step on the image using the learning rate. You should use
#
        # the grad variable you defined above.
#
        #
#
        # Be very careful about the signs of elements in your code.
#
        ################################################################################
##
        dx = sess.run(grad, feed_dict={model.image: X})[0]
        X = X + learning_rate * dx/np.linalg.norm(dx)
        pass
        ################################################################################
######
        #                             END OF YOUR CODE
#
        ################################################################################
######

        # Undo the jitter
        X = np.roll(np.roll(X, -ox, 1), -oy, 2)

        # As a regularizer, clip and periodically blur
        X = np.clip(X, -SQUEEZENET_MEAN/SQUEEZENET_STD, (1.0 - SQUEEZENET_MEA
N)/SQUEEZENET_STD)
        if t % blur_every == 0:
            X = blur_image(X, sigma=0.5)

        # Periodically show the image
        if t == 0 or (t + 1) % show_every == 0 or t == num_iterations - 1:
            plt.imshow(deprocess_image(X[0]))
            class_name = class_names[target_y]
            plt.title('%s\nIteration %d / %d' % (class_name, t + 1, num_iterat
ions))
            plt.gcf().set_size_inches(4, 4)
            plt.axis('off')
            plt.show()
    return X
```

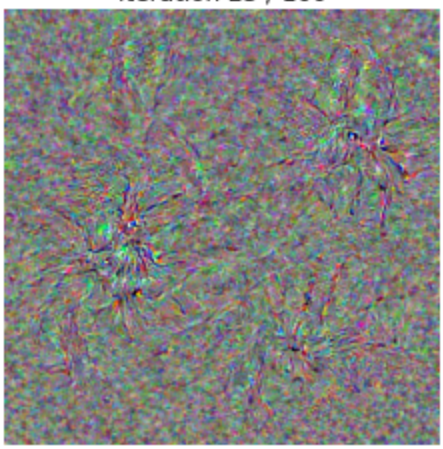
Once you have completed the implementation in the cell above, run the following cell to generate an image of Tarantula:

In [24]:
```
target_y = 76 # Tarantula
out = create_class_visualization(target_y, model)
```

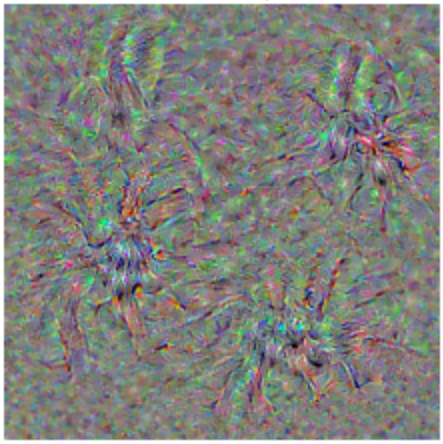
tarantula
Iteration 1 / 100



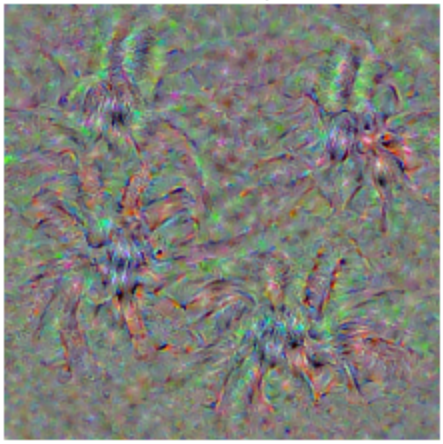
tarantula
Iteration 25 / 100



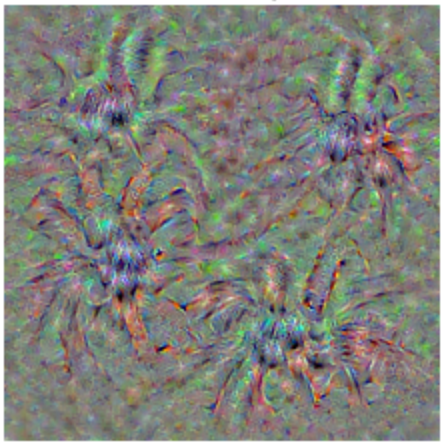
tarantula
Iteration 50 / 100



tarantula
Iteration 75 / 100



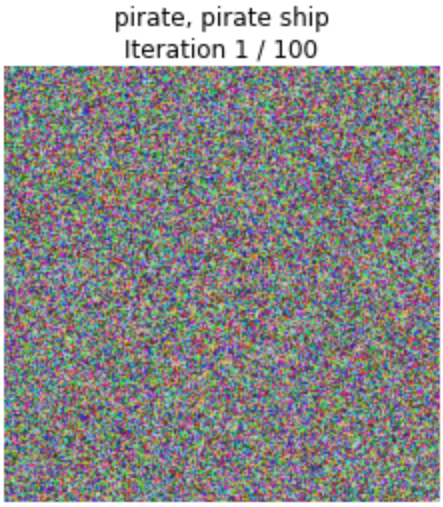
tarantula
Iteration 100 / 100

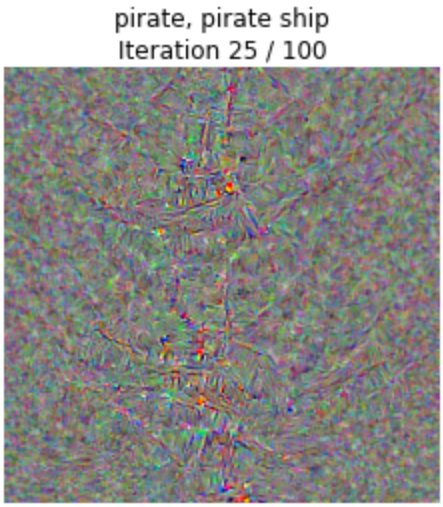


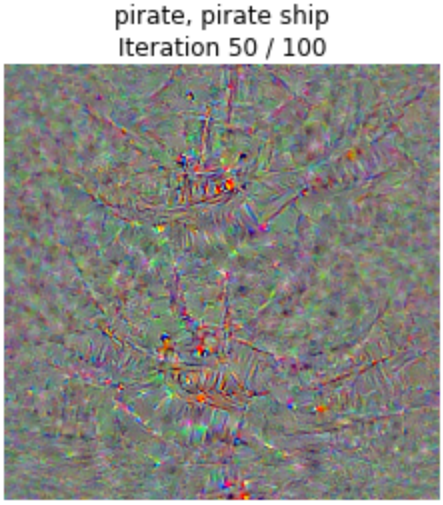
Try out your class visualization on other classes! You should also feel free to play with various hyperparameters to try and improve the quality of the generated image, but this is not required.

In [25]:
```
target_y = np.random.randint(1000)
# target_y = 78 # Tick
# target_y = 187 # Yorkshire Terrier
# target_y = 683 # Oboe
# target_y = 366 # Gorilla
# target_y = 604 # Hourglass
print(class_names[target_y])
X = create_class_visualization(target_y, model)
```

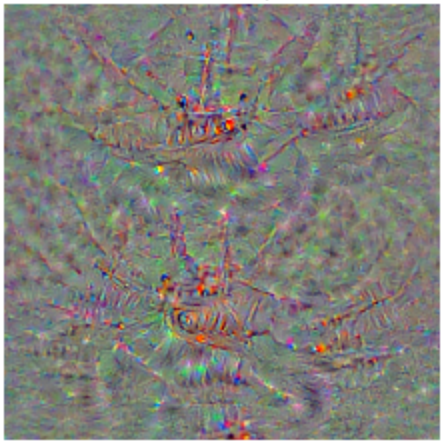
```
pirate, pirate ship
```



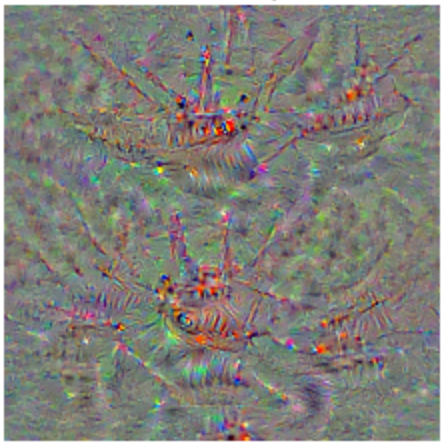




pirate, pirate ship
Iteration 75 / 100



pirate, pirate ship
Iteration 100 / 100



In [ ]:

In [ ]:
```
#COMMENT IF NOT USING COLAB VM

# This mounts your Google Drive to the Colab VM.
#from google.colab import drive
#drive.mount('/content/drive')

# TODO: Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'DeepLearning/assignments/assignment3/'
#FOLDERNAME = None
#assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
#import sys
#sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# This downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
#%cd /content/drive/My\ Drive/$FOLDERNAME/cs6353/datasets/
#!bash get_datasets.sh
#%cd /content/drive/My\ Drive/$FOLDERNAME
```

In [ ]:
```
# #UNCOMMENT IF USING CADE
# import os
# ##### Request a GPU #####
# ## This function locates an available gpu for usage. In addition, this funct
ion reserves a specificed
# ## memory space exclusively for your account. The memory reservation prevent
s the decrement in computational
# ## speed when other users try to allocate memory on the same gpu in the shar
ed systems, i.e., CADE machines.
# ## Note: If you use your own system which has a GPU with less than 4GB of me
mory, remember to change the
# ## specified mimimum memory.
# def define_gpu_to_use(minimum_memory_mb = 3500):
#     thres_memory = 600 #
#     gpu_to_use = None
#     try:
#         os.environ['CUDA_VISIBLE_DEVICES']
#         print('GPU already assigned before: ' + str(os.environ['CUDA_VISIBLE
_DEVICES']))
#         return
#     except:
#         pass

#     for i in range(16):
#         free_memory = !nvidia-smi --query-gpu=memory.free -i $i --format=cs
v,nounits,noheader
#         if free_memory[0] == 'No devices were found':
#             break
#         free_memory = int(free_memory[0])

#         if free_memory>minimum_memory_mb-thres_memory:
#             gpu_to_use = i
#             break

#     if gpu_to_use is None:
#         print('Could not find any GPU available with the required free memor
y of ' + str(minimum_memory_mb) \
#               + 'MB. Please use a different system for this assignment.')
#     else:
#         os.environ['CUDA_VISIBLE_DEVICES'] = str(gpu_to_use)
#         print('Chosen GPU: ' + str(gpu_to_use))

# ## Request a gpu and reserve the memory space
# define_gpu_to_use(4000)
```

Generative Adversarial Networks (GANs)

So far in cs6353, all the applications of neural networks that we have explored have been **discriminative models** that take an input and are trained to produce a labeled output. This has ranged from straightforward classification of image categories to sentence generation (which was still phrased as a classification problem, our labels were in vocabulary space and we'd learned a recurrence to capture multi-word labels). In this notebook, we will expand our repetoire, and build **generative models** using neural networks. Specifically, we will learn how to build models which generate novel images that resemble a set of training images.

What is a GAN?

In 2014, Goodfellow et al. (https://arxiv.org/abs/1406.2661) presented a method for training generative models called Generative Adversarial Networks (GANs for short). In a GAN, we build two different neural networks. Our first network is a traditional classification network, called the **discriminator**. We will train the discriminator to take images, and classify them as being real (belonging to the training set) or fake (not present in the training set). Our other network, called the **generator**, will take random noise as input and transform it using a neural network to produce images. The goal of the generator is to fool the discriminator into thinking the images it produced are real.

We can think of this back and forth process of the generator (G) trying to fool the discriminator (D), and the discriminator trying to correctly classify real vs. fake as a minimax game:

$$\underset{G}{\text{minimize}}\ \underset{D}{\text{maximize}}\ \mathbb{E}_{x \sim p_{\text{data}}}\left[\log D(x)\right] + \mathbb{E}_{z \sim p(z)}\left[\log(1 - D(G(z)))\right]$$

where $x \sim p_{\text{data}}$ are samples from the input data, $z \sim p(z)$ are the random noise samples, $G(z)$ are the generated images using the neural network generator G, and D is the output of the discriminator, specifying the probability of an input being real. In Goodfellow et al. (https://arxiv.org/abs/1406.2661), they analyze this minimax game and show how it relates to minimizing the Jensen-Shannon divergence between the training data distribution and the generated samples from G.

To optimize this minimax game, we will aternate between taking gradient *descent* steps on the objective for G, and gradient *ascent* steps on the objective for D:

1. update the **generator** (G) to minimize the probability of the **discriminator making the correct choice**.
2. update the **discriminator** (D) to maximize the probability of the **discriminator making the correct choice**.

While these updates are useful for analysis, they do not perform well in practice. Instead, we will use a different objective when we update the generator: maximize the probability of the **discriminator making the incorrect choice**. This small change helps to allevaiate problems with the generator gradient vanishing when the discriminator is confident. This is the standard update used in most GAN papers, and was used in the original paper from Goodfellow et al. (https://arxiv.org/abs/1406.2661).

In this assignment, we will alternate the following updates:

1. Update the generator (G) to maximize the probability of the discriminator making the incorrect choice on generated data:
$$\underset{G}{\text{maximize}}\ \mathbb{E}_{z \sim p(z)}\left[\log D(G(z))\right]$$
2. Update the discriminator (D), to maximize the probability of the discriminator making the correct choice on real and generated data:

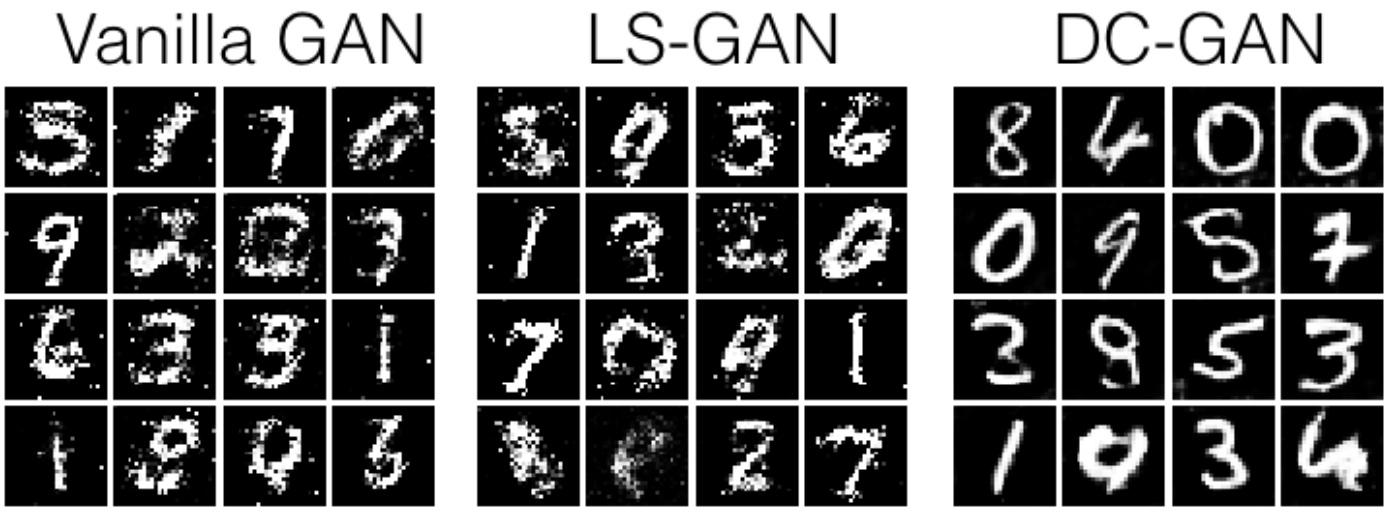
$$\underset{D}{\text{maximize}}\ \mathbb{E}_{x\sim p_{\text{data}}}\left[\log D(x)\right] + \mathbb{E}_{z\sim p(z)}\left[\log(1-D(G(z)))\right]$$

What else is there?

Since 2014, GANs have exploded into a huge research area, with massive workshops (https://sites.google.com/site/nips2016adversarial/), and hundreds of new papers (https://github.com/hindupuravinash/the-gan-zoo). Compared to other approaches for generative models, they often produce the highest quality samples but are some of the most difficult and finicky models to train (see this github repo (https://github.com/soumith/ganhacks) that contains a set of 17 hacks that are useful for getting models working). Improving the stabiilty and robustness of GAN training is an open research question, with new papers coming out every day! For a more recent tutorial on GANs, see here (https://arxiv.org/abs/1701.00160). There is also some even more recent exciting work that changes the objective function to Wasserstein distance and yields much more stable results across model architectures: WGAN (https://arxiv.org/abs/1701.07875), WGAN-GP (https://arxiv.org/abs/1704.00028).

GANs are not the only way to train a generative model! For other approaches to generative modeling check out the deep generative model chapter (http://www.deeplearningbook.org/contents/generative_models.html) of the Deep Learning book (http://www.deeplearningbook.org). Another popular way of training neural networks as generative models is Variational Autoencoders (co-discovered here (https://arxiv.org/abs/1312.6114) and here (https://arxiv.org/abs/1401.4082)). Variational autoencoders combine neural networks with variational inference to train deep generative models. These models tend to be far more stable and easier to train but currently don't produce samples that are as pretty as GANs.

Example pictures of what you should expect (yours might look slightly different):



Setup

In [1]:
```
import tensorflow.compat.v1 as tf
from tensorflow.python.framework import ops
import numpy as np
import os

import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'
tf.disable_v2_behavior()
# A bunch of utility functions

def show_images(images):
    images = np.reshape(images, [images.shape[0], -1])  # images reshape to (b
atch_size, D)
    sqrtn = int(np.ceil(np.sqrt(images.shape[0])))
    sqrtimg = int(np.ceil(np.sqrt(images.shape[1])))

    fig = plt.figure(figsize=(sqrtn, sqrtn))
    gs = gridspec.GridSpec(sqrtn, sqrtn)
    gs.update(wspace=0.05, hspace=0.05)

    for i, img in enumerate(images):
        ax = plt.subplot(gs[i])
        plt.axis('off')
        ax.set_xticklabels([])
        ax.set_yticklabels([])
        ax.set_aspect('equal')
        plt.imshow(img.reshape([sqrtimg,sqrtimg]))
    return

def preprocess_img(x):
    return 2 * x - 1.0

def deprocess_img(x):
    return (x + 1.0) / 2.0

def rel_error(x,y):
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))

def count_params():
    """Count the number of parameters in the current TensorFlow graph """
    param_count = np.sum([np.prod(x.get_shape().as_list()) for x in tf.global_
variables()])
    return param_count


def get_session():
    config = tf.ConfigProto()
```

```
    config.gpu_options.allow_growth = True
    session = tf.Session(config=config)
    return session

answers = np.load('gan-checks-tf.npz')
```

```
WARNING:tensorflow:From C:\Users\justi\anaconda3\envs\cs6353\lib\site-package
s\tensorflow\python\compat\v2_compat.py:101: disable_resource_variables (from
tensorflow.python.ops.variable_scope) is deprecated and will be removed in a
future version.
Instructions for updating:
non-resource variables are not supported in the long term
```

Dataset

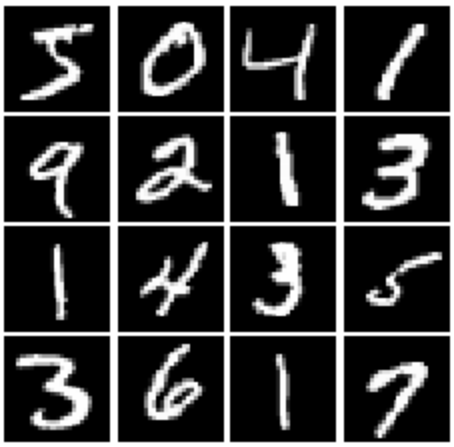
GANs are notoriously finicky with hyperparameters, and also require many training epochs. In order to make this assignment approachable without a GPU, we will be working on the MNIST dataset, which is 60,000 training and 10,000 test images. Each picture contains a centered image of white digit on black background (0 through 9). This was one of the first datasets used to train convolutional neural networks and it is fairly easy -- a standard CNN model can easily exceed 99% accuracy.

Heads-up: Our MNIST wrapper returns images as vectors. That is, they're size (batch, 784). If you want to treat them as images, we have to resize them to (batch,28,28) or (batch,28,28,1). They are also type np.float32 and bounded [0,1].

In [2]:
```
class MNIST(object):
    def __init__(self, batch_size, shuffle=False):
        """
        Construct an iterator object over the MNIST data

        Inputs:
        - batch_size: Integer giving number of elements per minibatch
        - shuffle: (optional) Boolean, whether to shuffle the data on each epo
ch
        """
        train, _ = tf.keras.datasets.mnist.load_data()
        X, y = train
        X = X.astype(np.float32)/255
        X = X.reshape((X.shape[0], -1))
        self.X, self.y = X, y
        self.batch_size, self.shuffle = batch_size, shuffle

    def __iter__(self):
        N, B = self.X.shape[0], self.batch_size
        idxs = np.arange(N)
        if self.shuffle:
            np.random.shuffle(idxs)
        return iter((self.X[i:i+B], self.y[i:i+B]) for i in range(0, N, B))
```

In [3]:
```
# show a batch
mnist = MNIST(batch_size=16)
show_images(mnist.X[:16])
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-data
sets/mnist.npz
11493376/11490434 [==============================] - 0s 0us/step
11501568/11490434 [==============================] - 0s 0us/step
```



LeakyReLU

In the cell below, you should implement a LeakyReLU. See the class notes (http://cs682.github.io/neural-networks-1/) (where alpha is small number) or equation (3) in this paper (http://ai.stanford.edu/~amaas/papers/relu_hybrid_icml2013_final.pdf). LeakyReLUs keep ReLU units from dying and are often used in GAN methods (as are maxout units, however those increase model size and therefore are not used in this notebook).

HINT: You should be able to use `tf.maximum`

In [6]:
```
def leaky_relu(x, alpha=0.01):
    """Compute the leaky ReLU activation function.

    Inputs:
    - x: TensorFlow Tensor with arbitrary shape
    - alpha: leak parameter for leaky ReLU

    Returns:
    TensorFlow Tensor with the same shape as x
    """
    # TODO: implement leaky ReLU
    return tf.maximum(alpha * x, x)
    pass
```

Test your leaky ReLU implementation. You should get errors < 1e-10

In [7]:
```
def test_leaky_relu(x, y_true):
    tf.reset_default_graph()
    with get_session() as sess:
        y_tf = leaky_relu(tf.constant(x))
        y = sess.run(y_tf)
        print('Maximum error: %g'%rel_error(y_true, y))

test_leaky_relu(answers['lrelu_x'], answers['lrelu_y'])
```

```
Maximum error: 0
```

Random Noise

Generate a TensorFlow `Tensor` containing uniform noise from -1 to 1 with shape `[batch_size, dim]`.

In [8]:
```
def sample_noise(batch_size, dim):
    """Generate random uniform noise from -1 to 1.

    Inputs:
    - batch_size: integer giving the batch size of noise to generate
    - dim: integer giving the dimension of the the noise to generate

    Returns:
    TensorFlow Tensor containing uniform noise in [-1, 1] with shape [batch_si
ze, dim]
    """
    # TODO: sample and return noise
    return tf.random_uniform([batch_size, dim], -1, 1)
    pass
```

Make sure noise is the correct shape and type:

In [9]:
```
def test_sample_noise():
    batch_size = 3
    dim = 4
    tf.reset_default_graph()
    with get_session() as sess:
        z = sample_noise(batch_size, dim)
        # Check z has the correct shape
        assert z.get_shape().as_list() == [batch_size, dim]
        # Make sure z is a Tensor and not a numpy array
        assert isinstance(z, tf.Tensor)
        # Check that we get different noise for different evaluations
        z1 = sess.run(z)
        z2 = sess.run(z)
        assert not np.array_equal(z1, z2)
        # Check that we get the correct range
        assert np.all(z1 >= -1.0) and np.all(z1 <= 1.0)
        print("All tests passed!")

test_sample_noise()
```

```
All tests passed!
```

Discriminator

Our first step is to build a discriminator. You should use the layers in `tf.layers` to build the model. All fully connected layers should include bias terms. For initialization, just use the default initializer used by the `tf.layers` functions.

Architecture:

- Fully connected layer with input size 784 and output size 256
- LeakyReLU with alpha 0.01
- Fully connected layer with output size 256
- LeakyReLU with alpha 0.01
- Fully connected layer with output size 1

The output of the discriminator should thus have shape `[batch_size, 1]`, and contain real numbers corresponding to the scores that each of the `batch_size` inputs is a real image.

In [12]:
```
def discriminator(x):
    """Compute discriminator score for a batch of input images.

    Inputs:
    - x: TensorFlow Tensor of flattened input images, shape [batch_size, 784]

    Returns:
    TensorFlow Tensor with shape [batch_size, 1], containing the score
    for an image being real for each input image.
    """
    with tf.variable_scope("discriminator"):
        # TODO: implement architecture
        fc = tf.layers.dense(x, 256)
        lr = leaky_relu(fc)
        fc2 = tf.layers.dense(lr, 256)
        lr2 = leaky_relu(fc2)
        logits = tf.layers.dense(lr2, 1)
        pass
        return logits
```

Test to make sure the number of parameters in the discriminator is correct:

In [13]:
```
def test_discriminator(true_count=267009):
    tf.reset_default_graph()
    with get_session() as sess:
        y = discriminator(tf.ones((2, 784)))
        cur_count = count_params()
        if cur_count != true_count:
            print('Incorrect number of parameters in discriminator. {0} instea
d of {1}. Check your achitecture.'.format(cur_count,true_count))
        else:
            print('Correct number of parameters in discriminator.')

test_discriminator()
```

```
Correct number of parameters in discriminator.
```

Generator

Now to build a generator. You should use the layers in `tf.layers` to construct the model. All fully connected layers should include bias terms. Note that you can use the tf.nn module to access activation functions. Once again, use the default initializers for parameters.

Architecture:

- Fully connected layer with inupt size tf.shape(z)[1] (the number of noise dimensions) and output size 1024
- `ReLU`
- Fully connected layer with output size 1024
- `ReLU`
- Fully connected layer with output size 784
- `TanH` (To restrict every element of the output to be in the range [-1,1])

In [14]:
```
def generator(z):
    """Generate images from a random noise vector.

    Inputs:
    - z: TensorFlow Tensor of random noise with shape [batch_size, noise_dim]

    Returns:
    TensorFlow Tensor of generated images, with shape [batch_size, 784].
    """
    with tf.variable_scope("generator"):
        # TODO: implement architecture
        fc1 = tf.layers.dense(z, 1024, activation = tf.nn.relu)
        fc2 = tf.layers.dense(fc1, 1024, activation = tf.nn.relu)
        img = tf.layers.dense(fc2, 784, activation = tf.nn.tanh)
        pass
        return img
```

Test to make sure the number of parameters in the generator is correct:

In [15]:
```
def test_generator(true_count=1858320):
    tf.reset_default_graph()
    with get_session() as sess:
        y = generator(tf.ones((1, 4)))
        cur_count = count_params()
        if cur_count != true_count:
            print('Incorrect number of parameters in generator. {0} instead of
{1}. Check your achitecture.'.format(cur_count,true_count))
        else:
            print('Correct number of parameters in generator.')

test_generator()
```

```
Correct number of parameters in generator.
```

GAN Loss

Compute the generator and discriminator loss. The generator loss is:

$$\ell_G = -\mathbb{E}_{z \sim p(z)}\left[\log D(G(z))\right]$$

and the discriminator loss is:

$$\ell_D = -\mathbb{E}_{x \sim p_{\text{data}}}\left[\log D(x)\right] - \mathbb{E}_{z \sim p(z)}\left[\log(1 - D(G(z)))\right]$$

Note that these are negated from the equations presented earlier as we will be *minimizing* these losses.

HINTS: Use tf.ones_like (https://www.tensorflow.org/api_docs/python/tf/ones_like) and tf.zeros_like (https://www.tensorflow.org/api_docs/python/tf/zeros_like) to generate labels for your discriminator. Use tf.nn.sigmoid_cross_entropy_with_logits (https://www.tensorflow.org/api_docs/python/tf/nn/sigmoid_cross_entropy_with_logits) to help compute your loss function. Instead of computing the expectation, we will be averaging over elements of the minibatch, so make sure to combine the loss by averaging instead of summing.

In [20]:
```
def gan_loss(logits_real, logits_fake):
    """Compute the GAN loss.

    Inputs:
    - logits_real: Tensor, shape [batch_size, 1], output of discriminator
        Unnormalized score that the image is real for each real image
    - logits_fake: Tensor, shape[batch_size, 1], output of discriminator
        Unnormalized score that the image is real for each fake image

    Returns:
    - D_loss: discriminator loss scalar
    - G_loss: generator loss scalar

    HINT: for the discriminator loss, you'll want to do the averaging separate
ly for
    its two components, and then add them together (instead of averaging once
at the very end).
    """
    # TODO: compute D_loss and G_loss
    Dx = tf.nn.sigmoid_cross_entropy_with_logits(logits = logits_real, labels
= tf.ones_like(logits_real))
    DGx = tf.nn.sigmoid_cross_entropy_with_logits(logits = logits_fake, labels
= tf.zeros_like(logits_fake))
    Gx = tf.nn.sigmoid_cross_entropy_with_logits(logits = logits_fake, labels
= tf.ones_like(logits_fake))
    D_loss = tf.reduce_mean(Dx) + tf.reduce_mean(DGx)
    G_loss = tf.reduce_mean(Gx)
    pass
    return D_loss, G_loss
```

Test your GAN loss. Make sure both the generator and discriminator loss are correct. You should see errors less than 1e-5.

In [21]:
```
def test_gan_loss(logits_real, logits_fake, d_loss_true, g_loss_true):
    tf.reset_default_graph()
    with get_session() as sess:
        d_loss, g_loss = sess.run(gan_loss(tf.constant(logits_real), tf.consta
nt(logits_fake)))
    print("Maximum error in d_loss: %g"%rel_error(d_loss_true, d_loss))
    print("Maximum error in g_loss: %g"%rel_error(g_loss_true, g_loss))

test_gan_loss(answers['logits_real'], answers['logits_fake'],
              answers['d_loss_true'], answers['g_loss_true'])
```

```
Maximum error in d_loss: 0
Maximum error in g_loss: 7.19722e-17
```

Optimizing our loss

Make an `AdamOptimizer` with a 1e-3 learning rate, beta1=0.5 to mininize G_loss and D_loss separately. The trick of decreasing beta was shown to be effective in helping GANs converge in the [Improved Techniques for Training GANs (https://arxiv.org/abs/1606.03498)](https://arxiv.org/abs/1606.03498) paper. In fact, with our current hyperparameters, if you set beta1 to the Tensorflow default of 0.9, there's a good chance your discriminator loss will go to zero and the generator will fail to learn entirely. In fact, this is a common failure mode in GANs; if your D(x) learns to be too fast (e.g. loss goes near zero), your G(z) is never able to learn. Often D(x) is trained with SGD with Momentum or RMSProp instead of Adam, but here we'll use Adam for both D(x) and G(z).

In [22]:
```
# TODO: create an AdamOptimizer for D_solver and G_solver
def get_solvers(learning_rate=1e-3, beta1=0.5):
    """Create solvers for GAN training.

    Inputs:
    - learning_rate: learning rate to use for both solvers
    - beta1: beta1 parameter for both solvers (first moment decay)

    Returns:
    - D_solver: instance of tf.train.AdamOptimizer with correct learning_rate
and beta1
    - G_solver: instance of tf.train.AdamOptimizer with correct learning_rate
and beta1
    """
    D_solver = tf.train.AdamOptimizer(learning_rate=learning_rate, beta1=beta
1)
    G_solver = tf.train.AdamOptimizer(learning_rate=learning_rate, beta1=beta
1)
    pass
    return D_solver, G_solver
```

Putting it all together

Now just a bit of Lego Construction.. Read this section over carefully to understand how we'll be composing the generator and discriminator

In [23]:
```
tf.reset_default_graph()

# number of images for each batch
batch_size = 128
# our noise dimension
noise_dim = 96

# placeholder for images from the training dataset
x = tf.placeholder(tf.float32, [None, 784])
# random noise fed into our generator
z = sample_noise(batch_size, noise_dim)
# generated images
G_sample = generator(z)

with tf.variable_scope("") as scope:
    #scale images to be -1 to 1
    logits_real = discriminator(preprocess_img(x))
    # Re-use discriminator weights on new inputs
    scope.reuse_variables()
    logits_fake = discriminator(G_sample)

# Get the list of variables for the discriminator and generator
D_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, 'discriminator')
G_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES, 'generator')

# get our solver
D_solver, G_solver = get_solvers()

# get our loss
D_loss, G_loss = gan_loss(logits_real, logits_fake)

# setup training steps
D_train_step = D_solver.minimize(D_loss, var_list=D_vars)
G_train_step = G_solver.minimize(G_loss, var_list=G_vars)
D_extra_step = tf.get_collection(tf.GraphKeys.UPDATE_OPS, 'discriminator')
G_extra_step = tf.get_collection(tf.GraphKeys.UPDATE_OPS, 'generator')
```

Training a GAN!

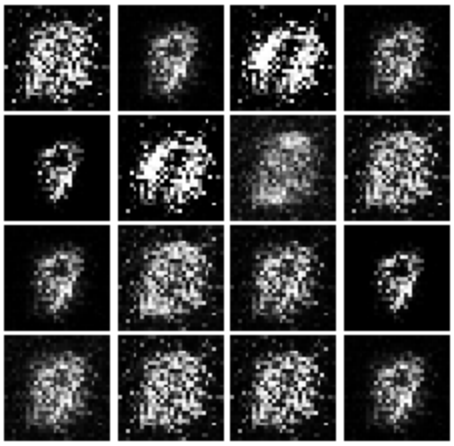
Well that wasn't so hard, was it? After the first epoch, you should see fuzzy outlines, clear shapes as you approach epoch 3, and decent shapes, about half of which will be sharp and clearly recognizable as we pass epoch 5. In our case, we'll simply train D(x) and G(z) with one batch each every iteration. However, papers often experiment with different schedules of training D(x) and G(z), sometimes doing one for more steps than the other, or even training each one until the loss gets "good enough" and then switching to training the other.

In [24]:
```
# a giant helper function
def run_a_gan(sess, G_train_step, G_loss, D_train_step, D_loss, G_extra_step,
D_extra_step,\
              show_every=2, print_every=1, batch_size=128, num_epoch=10):
    """Train a GAN for a certain number of epochs.

    Inputs:
    - sess: A tf.Session that we want to use to run our data
    - G_train_step: A training step for the Generator
    - G_loss: Generator loss
    - D_train_step: A training step for the Generator
    - D_loss: Discriminator loss
    - G_extra_step: A collection of tf.GraphKeys.UPDATE_OPS for generator
    - D_extra_step: A collection of tf.GraphKeys.UPDATE_OPS for discriminator
    Returns:
        Nothing
    """
    # compute the number of iterations we need
    mnist = MNIST(batch_size=batch_size, shuffle=True)
    for epoch in range(num_epoch):
        # every show often, show a sample result
        if epoch % show_every == 0:
            samples = sess.run(G_sample)
            fig = show_images(samples[:16])
            plt.show()
            print()
        for (minibatch, minbatch_y) in mnist:
            # run a batch of data through the network
            _, D_loss_curr = sess.run([D_train_step, D_loss], feed_dict={x: mi
nibatch})
            _, G_loss_curr = sess.run([G_train_step, G_loss])

        # print loss every so often.
        # We want to make sure D_loss doesn't go to 0
        if epoch % print_every == 0:
            print('Epoch: {}, D: {:.4}, G:{:.4}'.format(epoch,D_loss_curr,G_lo
ss_curr))
    print('Final images')
    samples = sess.run(G_sample)

    fig = show_images(samples[:16])
    plt.show()
```

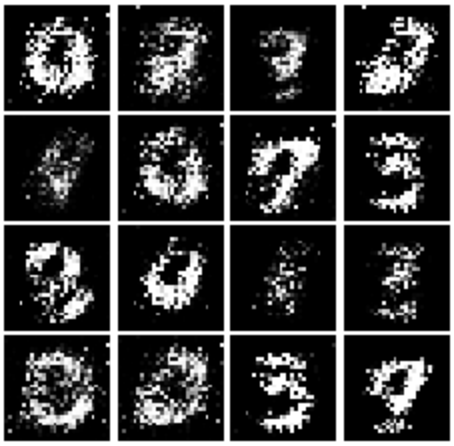
Train your GAN! This should take about 10 minutes on a CPU, or less than a minute on GPU.

In [25]:
```
with get_session() as sess:
    sess.run(tf.global_variables_initializer())
    run_a_gan(sess,G_train_step,G_loss,D_train_step,D_loss,G_extra_step,D_extr
a_step)
```

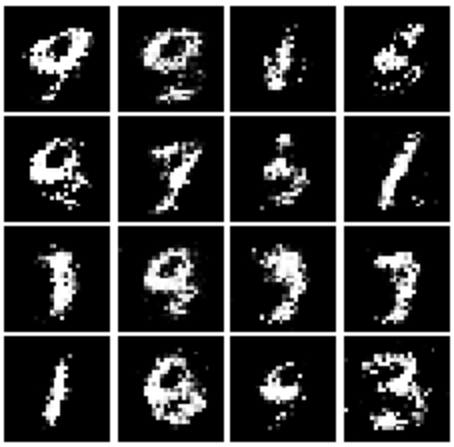
```
Epoch: 0, D: 1.656, G:0.934
Epoch: 1, D: 1.494, G:1.464
```



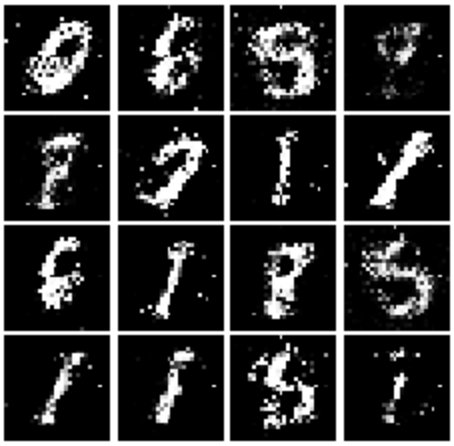
```
Epoch: 2, D: 1.547, G:0.7813
Epoch: 3, D: 1.423, G:0.8192
```



```
Epoch: 4, D: 1.458, G:0.8727
Epoch: 5, D: 1.483, G:0.8472
```



```
Epoch: 6, D: 1.444, G:0.868
Epoch: 7, D: 1.314, G:0.8728
```



```
Epoch: 8, D: 1.326, G:0.8122
Epoch: 9, D: 1.371, G:0.7581
Final images
```

Least Squares GAN

We'll now look at Least Squares GAN (https://arxiv.org/abs/1611.04076), a newer, more stable alternative to the original GAN loss function. For this part, all we have to do is change the loss function and retrain the model. We'll implement equation (9) in the paper, with the generator loss:

$$\ell_G = \frac{1}{2}\mathbb{E}_{z\sim p(z)}\left[\left(D(G(z))-1\right)^2\right]$$

and the discriminator loss:

$$\ell_D = \frac{1}{2}\mathbb{E}_{x\sim p_{\text{data}}}\left[\left(D(x)-1\right)^2\right] + \frac{1}{2}\mathbb{E}_{z\sim p(z)}\left[\left(D(G(z))\right)^2\right]$$

HINTS: Instead of computing the expectation, we will be averaging over elements of the minibatch, so make sure to combine the loss by averaging instead of summing. When plugging in for $D(x)$ and $D(G(z))$ use the direct output from the discriminator (`score_real` and `score_fake`).

In [26]:
```
def lsgan_loss(scores_real, scores_fake):
    """Compute the Least Squares GAN loss.

    Inputs:
    - scores_real: Tensor, shape [batch_size, 1], output of discriminator
        The score for each real image
    - scores_fake: Tensor, shape[batch_size, 1], output of discriminator
        The score for each fake image

    Returns:
    - D_loss: discriminator loss scalar
    - G_loss: generator loss scalar
    """
    # TODO: compute D_loss and G_loss
    D_loss = .5 * tf.reduce_mean((scores_real -1) ** 2) + .5 * tf.reduce_mean
(scores_fake ** 2)
    G_loss = .5 * tf.reduce_mean((scores_fake - 1) ** 2)
    pass
    return D_loss, G_loss
```

Test your LSGAN loss. You should see errors less than 1e-7.

In [27]:
```
def test_lsgan_loss(score_real, score_fake, d_loss_true, g_loss_true):
    with get_session() as sess:
        d_loss, g_loss = sess.run(
            lsgan_loss(tf.constant(score_real), tf.constant(score_fake)))
    print("Maximum error in d_loss: %g"%rel_error(d_loss_true, d_loss))
    print("Maximum error in g_loss: %g"%rel_error(g_loss_true, g_loss))

test_lsgan_loss(answers['logits_real'], answers['logits_fake'],
                answers['d_loss_lsgan_true'], answers['g_loss_lsgan_true'])
```

```
Maximum error in d_loss: 0
Maximum error in g_loss: 0
```

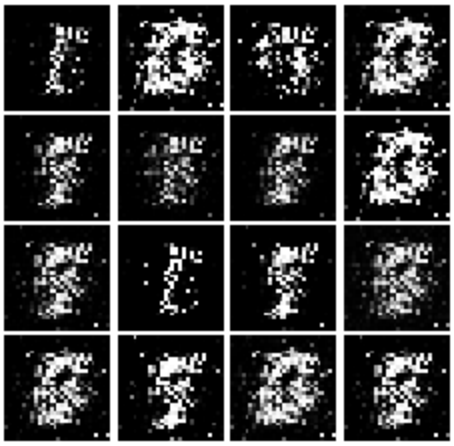
Create new training steps so we instead minimize the LSGAN loss:

In [28]:
```
D_loss, G_loss = lsgan_loss(logits_real, logits_fake)
D_train_step = D_solver.minimize(D_loss, var_list=D_vars)
G_train_step = G_solver.minimize(G_loss, var_list=G_vars)
```

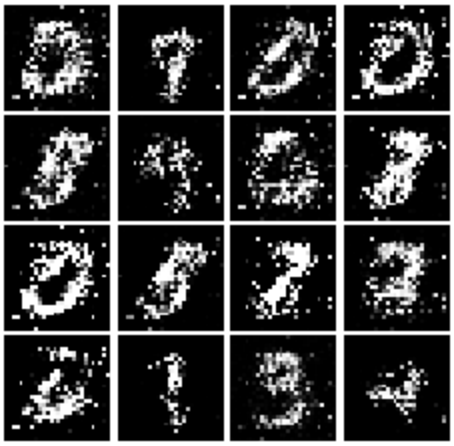
Run the following cell to train your model!

In [29]:
```
with get_session() as sess:
    sess.run(tf.global_variables_initializer())
    run_a_gan(sess, G_train_step, G_loss, D_train_step, D_loss, G_extra_step,
D_extra_step)
```

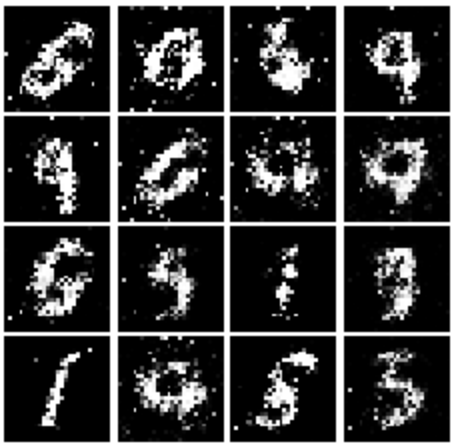
```
Epoch: 0, D: 0.04193, G:0.5418
Epoch: 1, D: 0.1125, G:0.297
```



```
Epoch: 2, D: 0.1181, G:0.3223
Epoch: 3, D: 0.1782, G:0.258
```

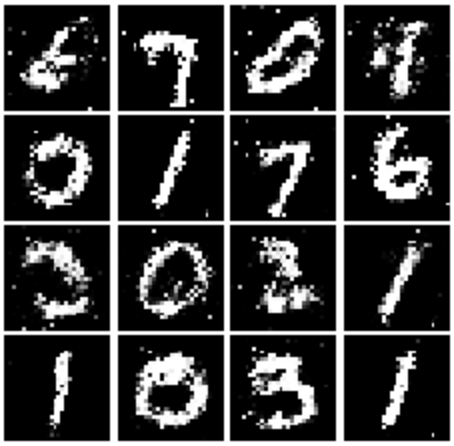


```
Epoch: 4, D: 0.2526, G:0.1795
Epoch: 5, D: 0.2097, G:0.2043
```



```
Epoch: 6, D: 0.1989, G:0.2109
Epoch: 7, D: 0.2358, G:0.1781
```

```
Epoch: 8, D: 0.2205, G:0.1739
Epoch: 9, D: 0.2395, G:0.1885
Final images
```



Deep Convolutional GANs

In the first part of the notebook, we implemented an almost direct copy of the original GAN network from Ian Goodfellow. However, this network architecture allows no real spatial reasoning. It is unable to reason about things like "sharp edges" in general because it lacks any convolutional layers. Thus, in this section, we will implement some of the ideas from DCGAN (https://arxiv.org/abs/1511.06434), where we use convolutional networks as our discriminators and generators.

Discriminator

We will use a discriminator inspired by the TensorFlow MNIST classification tutorial (https://www.tensorflow.org/get_started/mnist/pros), which is able to get above 99% accuracy on the MNIST dataset fairly quickly. *Be sure to check the dimensions of x and reshape when needed*, fully connected blocks expect [N,D] Tensors while conv2d blocks expect [N,H,W,C] Tensors. Please use `tf.layers` to define the following architecture:

Architecture:

- Conv2D: 32 Filters, 5x5, Stride 1, padding 0
- Leaky ReLU(alpha=0.01)
- Max Pool 2x2, Stride 2
- Conv2D: 64 Filters, 5x5, Stride 1, padding 0
- Leaky ReLU(alpha=0.01)
- Max Pool 2x2, Stride 2
- Flatten
- Fully Connected with output size 4 x 4 x 64
- Leaky ReLU(alpha=0.01)
- Fully Connected with output size 1

Once again, please use biases for all convolutional and fully connected layers, and use the default parameter initializers. Note that a padding of 0 can be accomplished with the 'VALID' padding option.

In [32]:
```
def discriminator(x):
    """Compute discriminator score for a batch of input images.

    Inputs:
    - x: TensorFlow Tensor of flattened input images, shape [batch_size, 784]

    Returns:
    TensorFlow Tensor with shape [batch_size, 1], containing the score
    for an image being real for each input image.
    """
    with tf.variable_scope("discriminator"):
        # TODO: implement architecture
        x = tf.reshape(x, [-1, 28, 28, 1])
        conv1 = tf.layers.conv2d(x, 32, 5, 1, padding = "valid", activation =
leaky_relu)
        pool1 = tf.layers.max_pooling2d(conv1, 2, 2)
        conv2 = tf.layers.conv2d(pool1, 64, 5, 1, padding = "valid", activatio
n = leaky_relu)
        pool2 = tf.layers.max_pooling2d(conv2, 2, 2)
        pool2 = tf.layers.flatten(pool2)
        fc1 = tf.layers.dense(pool2, 4 * 4 * 64, activation = leaky_relu)
        logits = tf.layers.dense(fc1, 1)
        pass
        return logits
test_discriminator(1102721)
```

```
Correct number of parameters in discriminator.
```

Generator

For the generator, we will copy the architecture exactly from the InfoGAN paper (https://arxiv.org/pdf/1606.03657.pdf). See Appendix C.1 MNIST. Please use `tf.layers` for your implementation. You might find the documentation for tf.layers.conv2d_transpose (https://www.tensorflow.org/api_docs/python/tf/layers/conv2d_transpose) useful. The architecture is as follows.

Architecture:

- Fully connected with output size 1024
- `ReLU`
- BatchNorm
- Fully connected with output size 7 x 7 x 128
- `ReLU`
- BatchNorm
- Resize into Image Tensor of size 7, 7, 128
- Conv2D^T (transpose): 64 filters of 4x4, stride 2
- `ReLU`
- BatchNorm
- Conv2d^T (transpose): 1 filter of 4x4, stride 2
- `TanH`

Once again, use biases for the fully connected and transpose convolutional layers. Please use the default initializers for your parameters. For padding, choose the 'same' option for transpose convolutions. For Batch Normalization, assume we are always in 'training' mode.

In [34]:
```
def generator(z):
    """Generate images from a random noise vector.

    Inputs:
    - z: TensorFlow Tensor of random noise with shape [batch_size, noise_dim]

    Returns:
    TensorFlow Tensor of generated images, with shape [batch_size, 784].
    """
    with tf.variable_scope("generator"):
        # TODO: implement architecture
        fc1 = tf.layers.dense(z, 1024, activation = tf.nn.relu)
        bn1 = tf.layers.batch_normalization(fc1, training = True)
        fc2 = tf.layers.dense(bn1, 7 * 7 * 128, activation = tf.nn.relu)
        bn2 = tf.layers.batch_normalization(fc2, training = True)
        out = tf.reshape(bn2, [-1, 7, 7, 128])
        conv1 = tf.layers.conv2d_transpose(out, 64, 4, 2, padding = "same", ac
tivation = tf.nn.relu)
        bn3 = tf.layers.batch_normalization(conv1, training = True)
        img = tf.layers.conv2d_transpose(bn3, 1, 4, 2, padding = "same", activ
ation = tf.nn.tanh)
        pass
        return img
test_generator(6595521)
```

```
Correct number of parameters in generator.
```

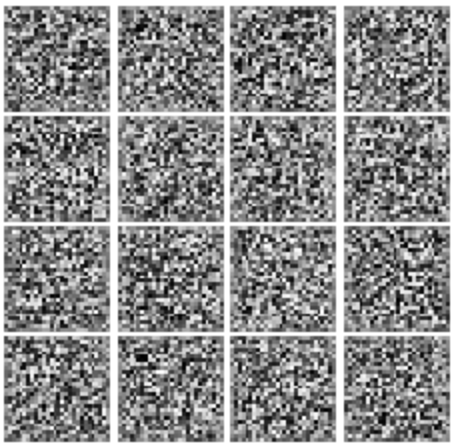
We have to recreate our network since we've changed our functions.

In [35]:
```
tf.reset_default_graph()

batch_size = 128
# our noise dimension
noise_dim = 96

# placeholders for images from the training dataset
x = tf.placeholder(tf.float32, [None, 784])
z = sample_noise(batch_size, noise_dim)
# generated images
G_sample = generator(z)

with tf.variable_scope("") as scope:
    #scale images to be -1 to 1
    logits_real = discriminator(preprocess_img(x))
    # Re-use discriminator weights on new inputs
    scope.reuse_variables()
    logits_fake = discriminator(G_sample)

# Get the list of variables for the discriminator and generator
D_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,'discriminator')
G_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,'generator')

D_solver,G_solver = get_solvers()
D_loss, G_loss = gan_loss(logits_real, logits_fake)
D_train_step = D_solver.minimize(D_loss, var_list=D_vars)
G_train_step = G_solver.minimize(G_loss, var_list=G_vars)
D_extra_step = tf.get_collection(tf.GraphKeys.UPDATE_OPS,'discriminator')
G_extra_step = tf.get_collection(tf.GraphKeys.UPDATE_OPS,'generator')
```

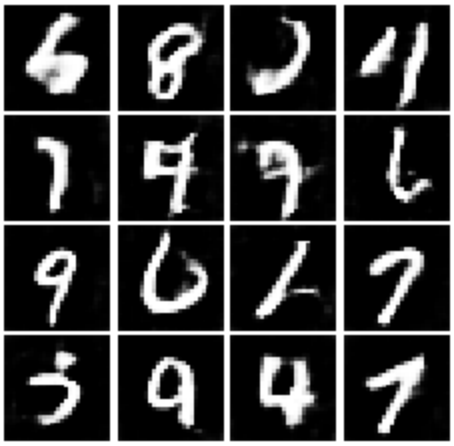
Train and evaluate a DCGAN

This is the one part of A3 that significantly benefits from using a GPU. It takes 3 minutes on a GPU for the requested five epochs. Or about 50 minutes on a dual core laptop on CPU (feel free to use 3 epochs if you do it on CPU).

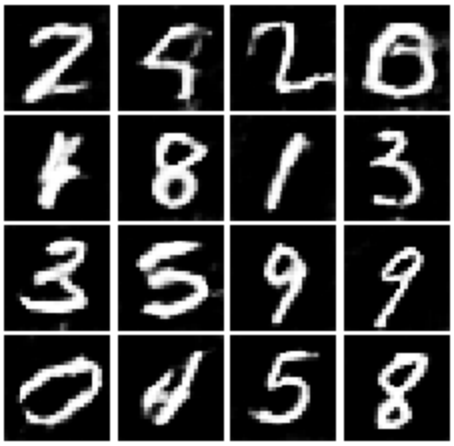
In [36]:
```
with get_session() as sess:
    sess.run(tf.global_variables_initializer())
    run_a_gan(sess,G_train_step,G_loss,D_train_step,D_loss,G_extra_step,D_extr
a_step,num_epoch=5)
```



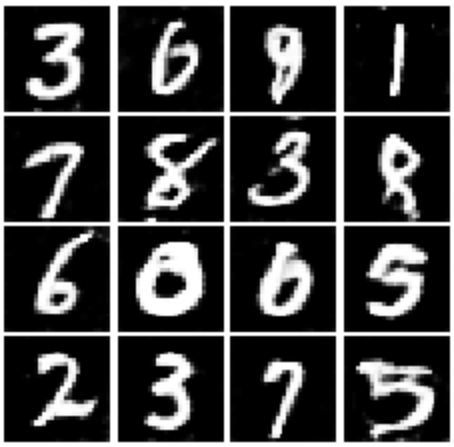
```
Epoch: 0, D: 1.037, G:1.136
Epoch: 1, D: 1.017, G:0.879
```



```
Epoch: 2, D: 1.075, G:0.845
Epoch: 3, D: 0.9813, G:0.9174
```



```
Epoch: 4, D: 1.011, G:0.7332
Final images
```



INLINE QUESTION 1

If the generator loss decreases during training while the discriminator loss stays at a constant high value from the start, is this a good sign? Why or why not? A qualitative answer is sufficient

Your answer:

It is not a good sign because it means that the discriminator is not learning to correctly classify real images. This means that it could classify noisy images as real ones. Since the generator starts with noisy images it would end up classifying them as real images which makes the generator loss decrease even if the images are noisy.