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 [1]:
```
# As usual, a bit of setup
from __future__ import print_function
import time, os, json
import numpy as np
import matplotlib.pyplot as plt

from cs231n.gradient_check import eval_numerical_gradient, eval_numerical_gradient_array
from cs231n.rnn_layers import *
from cs231n.captioning_solver import CaptioningSolver
from cs231n.classifiers.rnn import CaptioningRNN
from cs231n.coco_utils import load_coco_data, sample_coco_minibatch, decode_captions
from cs231n.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-ipython
%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))))
```

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. 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 [2]:
```
!pip install h5py
```

```
Requirement already satisfied: h5py in /miniconda3/envs/cs4803/lib/python3.6/site-packages (2.10.0)
Requirement already satisfied: numpy>=1.7 in /miniconda3/envs/cs4803/lib/python3.6/site-packages (from h5py) (1.18.1)
Requirement already satisfied: six in /miniconda3/envs/cs4803/lib/python3.6/site-packages (from h5py) (1.14.0)
```

Microsoft COCO

For this exercise we will use the 2014 release of the [Microsoft COCO dataset (http://mscoco.org/)](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 `cs231n/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 `cs231n/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 `cs231n/coco_utils.py`. Run the following cell to do so:

In [3]:
```python
# 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 `cs231n/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 [4]:
```python
# 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 small bathroom with a white toilet sitting next to a sink <END>

<START> a house with a two piece swinging door that is open <END>

<START> a woman and toddler sitting on a bench with <UNK> <END>

Recurrent Neural Networks

As discussed in lecture, we will use recurrent neural network (RNN) language models for image captioning. The file `cs231n/rnn_layers.py` contains implementations of different layer types that are needed for recurrent neural networks, and the file `cs231n/classifiers/rnn.py` uses these layers to implement an image captioning model.

We will first implement different types of RNN layers in `cs231n/rnn_layers.py`.

Vanilla RNN: step forward

Open the file `cs231n/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 less than 1e-8.

In [5]:
```
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 `cs231n/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 less than `1e-8` .

In [6]:
```
from cs231n.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))
```

```
N 4, D 5, H 6
dnext_h: (N, H) (4, 6)
x: (N, D) (4, 5)
prev_h: (N, H) (4, 6)
Wx: (D, H) (5, 6)
Wh: (H, H) (6, 6)
dTanH: (N, H) (4, 6)
dSum: (N, H) (4, 6)
dx: (4, 5)
dprev_h: (4, 6)
dWx: (5, 6)
dWh: (6, 6)
db: (6,)
dx error:  2.99311613693832e-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 process an entire sequence of data.

In the file `cs231n/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 less than `1e-7` .

In [7]:
```
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 `cs231n/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, calling into the `rnn_step_backward` function that you defined above. You should see errors less than 5e-7.

In [8]:
```
np.random.seed(231)

N, D, T, H = 2, 3, 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(f"correct dx: {dx_num}")
# print(f"reagans dx: {dx}")
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:  2.3969112188524054e-09
dh0 error:  3.3796875007867145e-09
dWx error:  7.221000108504998e-09
dWh error:  1.284586847530015e-07
db error:  4.675767378424171e-10
```

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 `cs231n/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 error around `1e-8` .

In [9]:
```
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 errors less than `1e-11` .

In [10]:
```
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 `cs231n/rnn_layers.py` . Run the following to perform numeric gradient checking on the implementation. You should see errors less than 1e-9.

In [11]:
```
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 `cs231n/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 less than 1e-7.

```
In [12]: # Sanity check for temporal softmax loss
from cs231n.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 `cs231n/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 less than `1e-10`.

```
In [13]: N, D, W, H = 10, 20, 30, 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, verbose=False)
expected_loss = 9.83235591003

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

```
loss:  9.832355910027388
expected loss:  9.83235591003
difference:  2.611244553918368e-12
```

Run the following cell to perform numeric gradient checking on the `CaptioningRNN` class; you should errors around `5e-6` or less.

In [14]:
```
np.random.seed(231)

batch_size = 2
timesteps = 3
input_dim = 4
wordvec_dim = 5
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], verbose=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: 2.331072e-09
W_proj relative error: 9.974424e-09
W_vocab relative error: 4.274378e-09
Wh relative error: 5.954804e-09
Wx relative error: 8.455229e-07
b relative error: 9.727211e-10
b_proj relative error: 1.991603e-08
b_vocab relative error: 6.918525e-11
```

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 `cs231n/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 overfit a small sample of 100 training examples. You should see losses of less than 0.1.

In [15]:
```
np.random.seed(231)

small_data = load_coco_data(max_train=100)#00)

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=1,#100
           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()
```

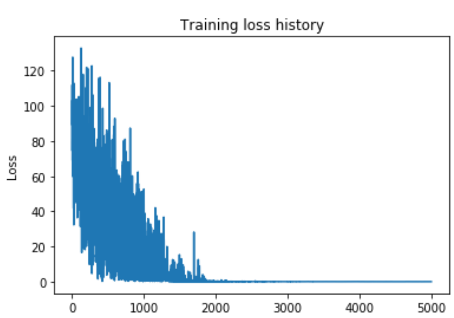
```
(Iteration 1 / 5000) loss: 89.538734
(Iteration 11 / 5000) loss: 89.436142
(Iteration 21 / 5000) loss: 41.782749
(Iteration 31 / 5000) loss: 97.063431
(Iteration 41 / 5000) loss: 98.784393
(Iteration 51 / 5000) loss: 103.743103
(Iteration 61 / 5000) loss: 76.610619
(Iteration 71 / 5000) loss: 46.232353
(Iteration 81 / 5000) loss: 47.529114
(Iteration 91 / 5000) loss: 51.454792
(Iteration 101 / 5000) loss: 41.202644
(Iteration 111 / 5000) loss: 103.156456
(Iteration 121 / 5000) loss: 63.173302
(Iteration 131 / 5000) loss: 37.913567
(Iteration 141 / 5000) loss: 99.907059
(Iteration 151 / 5000) loss: 85.212387
(Iteration 161 / 5000) loss: 57.529522
(Iteration 171 / 5000) loss: 21.788654
(Iteration 181 / 5000) loss: 27.034616
(Iteration 191 / 5000) loss: 29.954041
(Iteration 201 / 5000) loss: 20.125360
(Iteration 211 / 5000) loss: 42.703835
(Iteration 221 / 5000) loss: 50.779907
(Iteration 231 / 5000) loss: 61.426395
(Iteration 241 / 5000) loss: 64.845703
(Iteration 251 / 5000) loss: 39.612869
(Iteration 261 / 5000) loss: 45.992516
(Iteration 271 / 5000) loss: 14.378930
(Iteration 281 / 5000) loss: 22.156929
(Iteration 291 / 5000) loss: 9.530903
(Iteration 301 / 5000) loss: 21.537636
(Iteration 311 / 5000) loss: 28.899513
(Iteration 321 / 5000) loss: 71.319817
(Iteration 331 / 5000) loss: 14.083916
(Iteration 341 / 5000) loss: 45.902863
(Iteration 351 / 5000) loss: 9.437120
(Iteration 361 / 5000) loss: 54.087708
(Iteration 371 / 5000) loss: 16.939861
(Iteration 381 / 5000) loss: 66.815460
(Iteration 391 / 5000) loss: 30.158628
(Iteration 401 / 5000) loss: 55.309616
(Iteration 411 / 5000) loss: 2.467841
(Iteration 421 / 5000) loss: 16.460686
(Iteration 431 / 5000) loss: 45.130829
(Iteration 441 / 5000) loss: 53.272102
(Iteration 451 / 5000) loss: 51.743515
(Iteration 461 / 5000) loss: 33.269196
(Iteration 471 / 5000) loss: 51.398998
(Iteration 481 / 5000) loss: 35.818146
(Iteration 491 / 5000) loss: 15.107388
(Iteration 501 / 5000) loss: 54.070717
(Iteration 511 / 5000) loss: 62.900036
(Iteration 521 / 5000) loss: 41.764988
(Iteration 531 / 5000) loss: 12.469192
(Iteration 541 / 5000) loss: 60.341114
(Iteration 551 / 5000) loss: 25.075655
(Iteration 561 / 5000) loss: 6.382981
(Iteration 571 / 5000) loss: 2.519198
(Iteration 581 / 5000) loss: 40.610477
(Iteration 591 / 5000) loss: 53.783371
(Iteration 601 / 5000) loss: 10.248645
(Iteration 611 / 5000) loss: 4.130352
(Iteration 621 / 5000) loss: 24.409468
(Iteration 631 / 5000) loss: 16.780783
(Iteration 641 / 5000) loss: 32.698448
(Iteration 651 / 5000) loss: 8.323520
(Iteration 661 / 5000) loss: 48.345177
(Iteration 671 / 5000) loss: 55.929413
(Iteration 681 / 5000) loss: 37.345932
(Iteration 691 / 5000) loss: 47.278160
(Iteration 701 / 5000) loss: 9.752253
(Iteration 711 / 5000) loss: 17.875879
(Iteration 721 / 5000) loss: 80.255623
(Iteration 731 / 5000) loss: 5.435860
(Iteration 741 / 5000) loss: 27.480110
(Iteration 751 / 5000) loss: 14.788023
(Iteration 761 / 5000) loss: 13.484448
(Iteration 771 / 5000) loss: 40.191666
(Iteration 781 / 5000) loss: 28.038023
(Iteration 791 / 5000) loss: 17.346283
(Iteration 801 / 5000) loss: 31.922714
(Iteration 811 / 5000) loss: 24.746355
(Iteration 821 / 5000) loss: 0.960066
(Iteration 831 / 5000) loss: 3.729098
(Iteration 841 / 5000) loss: 24.331656
(Iteration 851 / 5000) loss: 1.427623
(Iteration 861 / 5000) loss: 4.385690
(Iteration 871 / 5000) loss: 19.940441
(Iteration 881 / 5000) loss: 47.771824
(Iteration 891 / 5000) loss: 7.549527
(Iteration 901 / 5000) loss: 34.210579
(Iteration 911 / 5000) loss: 7.658073
(Iteration 921 / 5000) loss: 4.147122
(Iteration 931 / 5000) loss: 7.015904
(Iteration 941 / 5000) loss: 0.634538
(Iteration 951 / 5000) loss: 2.414166
(Iteration 961 / 5000) loss: 3.767045
(Iteration 971 / 5000) loss: 9.693328
(Iteration 981 / 5000) loss: 0.422882
(Iteration 991 / 5000) loss: 15.993933
(Iteration 1001 / 5000) loss: 3.902276
(Iteration 1011 / 5000) loss: 14.404881
(Iteration 1021 / 5000) loss: 1.892224
(Iteration 1031 / 5000) loss: 9.150167
```

```
(Iteration 1041 / 5000) loss: 1.529371
(Iteration 1051 / 5000) loss: 9.478404
(Iteration 1061 / 5000) loss: 5.039266
(Iteration 1071 / 5000) loss: 12.174468
(Iteration 1081 / 5000) loss: 14.921804
(Iteration 1091 / 5000) loss: 6.668529
(Iteration 1101 / 5000) loss: 3.538936
(Iteration 1111 / 5000) loss: 9.991777
(Iteration 1121 / 5000) loss: 5.878150
(Iteration 1131 / 5000) loss: 0.295447
(Iteration 1141 / 5000) loss: 2.295105
(Iteration 1151 / 5000) loss: 0.180497
(Iteration 1161 / 5000) loss: 19.845589
(Iteration 1171 / 5000) loss: 3.390728
(Iteration 1181 / 5000) loss: 5.576392
(Iteration 1191 / 5000) loss: 6.027318
(Iteration 1201 / 5000) loss: 1.031552
(Iteration 1211 / 5000) loss: 0.574944
(Iteration 1221 / 5000) loss: 0.266748
(Iteration 1231 / 5000) loss: 0.517425
(Iteration 1241 / 5000) loss: 0.115634
(Iteration 1251 / 5000) loss: 0.857323
(Iteration 1261 / 5000) loss: 11.045198
(Iteration 1271 / 5000) loss: 26.033632
(Iteration 1281 / 5000) loss: 5.509893
(Iteration 1291 / 5000) loss: 5.609757
(Iteration 1301 / 5000) loss: 7.111456
(Iteration 1311 / 5000) loss: 0.598661
(Iteration 1321 / 5000) loss: 19.992622
(Iteration 1331 / 5000) loss: 2.082992
(Iteration 1341 / 5000) loss: 4.956313
(Iteration 1351 / 5000) loss: 0.214364
(Iteration 1361 / 5000) loss: 0.750437
(Iteration 1371 / 5000) loss: 0.400560
(Iteration 1381 / 5000) loss: 0.188203
(Iteration 1391 / 5000) loss: 0.574202
(Iteration 1401 / 5000) loss: 0.292207
(Iteration 1411 / 5000) loss: 0.089854
(Iteration 1421 / 5000) loss: 0.568982
(Iteration 1431 / 5000) loss: 3.127827
(Iteration 1441 / 5000) loss: 0.206117
(Iteration 1451 / 5000) loss: 0.121819
(Iteration 1461 / 5000) loss: 0.216600
(Iteration 1471 / 5000) loss: 0.105744
(Iteration 1481 / 5000) loss: 0.452972
(Iteration 1491 / 5000) loss: 0.099933
(Iteration 1501 / 5000) loss: 0.253767
(Iteration 1511 / 5000) loss: 0.397597
(Iteration 1521 / 5000) loss: 0.262964
(Iteration 1531 / 5000) loss: 0.636551
(Iteration 1541 / 5000) loss: 0.238412
(Iteration 1551 / 5000) loss: 5.855202
(Iteration 1561 / 5000) loss: 0.406963
(Iteration 1571 / 5000) loss: 0.165135
(Iteration 1581 / 5000) loss: 0.304915
(Iteration 1591 / 5000) loss: 0.599466
(Iteration 1601 / 5000) loss: 1.366495
(Iteration 1611 / 5000) loss: 0.090246
(Iteration 1621 / 5000) loss: 0.168135
(Iteration 1631 / 5000) loss: 0.187476
(Iteration 1641 / 5000) loss: 0.168081
(Iteration 1651 / 5000) loss: 0.712782
(Iteration 1661 / 5000) loss: 0.584735
(Iteration 1671 / 5000) loss: 0.069395
(Iteration 1681 / 5000) loss: 0.728173
(Iteration 1691 / 5000) loss: 0.069698
(Iteration 1701 / 5000) loss: 0.063079
(Iteration 1711 / 5000) loss: 0.041461
(Iteration 1721 / 5000) loss: 0.562342
(Iteration 1731 / 5000) loss: 0.072204
(Iteration 1741 / 5000) loss: 0.548008
(Iteration 1751 / 5000) loss: 0.038358
(Iteration 1761 / 5000) loss: 0.676519
(Iteration 1771 / 5000) loss: 0.033160
(Iteration 1781 / 5000) loss: 0.168352
(Iteration 1791 / 5000) loss: 0.272866
(Iteration 1801 / 5000) loss: 0.644750
(Iteration 1811 / 5000) loss: 0.466863
(Iteration 1821 / 5000) loss: 0.331481
(Iteration 1831 / 5000) loss: 0.051933
(Iteration 1841 / 5000) loss: 0.252053
(Iteration 1851 / 5000) loss: 0.093088
(Iteration 1861 / 5000) loss: 0.265629
(Iteration 1871 / 5000) loss: 0.222783
(Iteration 1881 / 5000) loss: 0.141227
(Iteration 1891 / 5000) loss: 0.115807
(Iteration 1901 / 5000) loss: 0.104958
(Iteration 1911 / 5000) loss: 0.066892
(Iteration 1921 / 5000) loss: 0.075094
(Iteration 1931 / 5000) loss: 0.064218
(Iteration 1941 / 5000) loss: 0.094453
(Iteration 1951 / 5000) loss: 0.270463
(Iteration 1961 / 5000) loss: 0.100966
(Iteration 1971 / 5000) loss: 0.096354
(Iteration 1981 / 5000) loss: 0.107349
(Iteration 1991 / 5000) loss: 0.160119
(Iteration 2001 / 5000) loss: 0.054516
(Iteration 2011 / 5000) loss: 0.125294
(Iteration 2021 / 5000) loss: 0.325757
(Iteration 2031 / 5000) loss: 0.077539
(Iteration 2041 / 5000) loss: 0.125360
(Iteration 2051 / 5000) loss: 0.080237
(Iteration 2061 / 5000) loss: 0.068496
(Iteration 2071 / 5000) loss: 0.072739
(Iteration 2081 / 5000) loss: 0.113197
```

```
(Iteration 2091 / 5000) loss: 0.080975
(Iteration 2101 / 5000) loss: 0.096208
(Iteration 2111 / 5000) loss: 0.059108
(Iteration 2121 / 5000) loss: 0.034324
(Iteration 2131 / 5000) loss: 0.032771
(Iteration 2141 / 5000) loss: 0.050986
(Iteration 2151 / 5000) loss: 0.033320
(Iteration 2161 / 5000) loss: 0.059138
(Iteration 2171 / 5000) loss: 0.049778
(Iteration 2181 / 5000) loss: 0.134106
(Iteration 2191 / 5000) loss: 0.132563
(Iteration 2201 / 5000) loss: 0.054139
(Iteration 2211 / 5000) loss: 0.102579
(Iteration 2221 / 5000) loss: 0.044399
(Iteration 2231 / 5000) loss: 0.027050
(Iteration 2241 / 5000) loss: 0.083712
(Iteration 2251 / 5000) loss: 0.048737
(Iteration 2261 / 5000) loss: 0.087930
(Iteration 2271 / 5000) loss: 0.065641
(Iteration 2281 / 5000) loss: 0.066378
(Iteration 2291 / 5000) loss: 0.134018
(Iteration 2301 / 5000) loss: 0.063139
(Iteration 2311 / 5000) loss: 0.049333
(Iteration 2321 / 5000) loss: 0.046218
(Iteration 2331 / 5000) loss: 0.062667
(Iteration 2341 / 5000) loss: 0.155200
(Iteration 2351 / 5000) loss: 0.077619
(Iteration 2361 / 5000) loss: 0.073163
(Iteration 2371 / 5000) loss: 0.038135
(Iteration 2381 / 5000) loss: 0.045802
(Iteration 2391 / 5000) loss: 0.179533
(Iteration 2401 / 5000) loss: 0.045145
(Iteration 2411 / 5000) loss: 0.046927
(Iteration 2421 / 5000) loss: 0.063020
(Iteration 2431 / 5000) loss: 0.040154
(Iteration 2441 / 5000) loss: 0.029081
(Iteration 2451 / 5000) loss: 0.092092
(Iteration 2461 / 5000) loss: 0.024319
(Iteration 2471 / 5000) loss: 0.052775
(Iteration 2481 / 5000) loss: 0.051961
(Iteration 2491 / 5000) loss: 0.043971
(Iteration 2501 / 5000) loss: 0.074047
(Iteration 2511 / 5000) loss: 0.037329
(Iteration 2521 / 5000) loss: 0.069169
(Iteration 2531 / 5000) loss: 0.049513
(Iteration 2541 / 5000) loss: 0.041850
(Iteration 2551 / 5000) loss: 0.429351
(Iteration 2561 / 5000) loss: 0.045865
(Iteration 2571 / 5000) loss: 0.051164
(Iteration 2581 / 5000) loss: 0.037816
(Iteration 2591 / 5000) loss: 0.028778
(Iteration 2601 / 5000) loss: 0.048968
(Iteration 2611 / 5000) loss: 0.060120
(Iteration 2621 / 5000) loss: 0.077502
(Iteration 2631 / 5000) loss: 0.068717
(Iteration 2641 / 5000) loss: 0.034594
(Iteration 2651 / 5000) loss: 0.185056
(Iteration 2661 / 5000) loss: 0.039449
(Iteration 2671 / 5000) loss: 0.063915
(Iteration 2681 / 5000) loss: 0.086589
(Iteration 2691 / 5000) loss: 0.153010
(Iteration 2701 / 5000) loss: 0.108605
(Iteration 2711 / 5000) loss: 0.043920
(Iteration 2721 / 5000) loss: 0.054018
(Iteration 2731 / 5000) loss: 0.036808
(Iteration 2741 / 5000) loss: 0.064860
(Iteration 2751 / 5000) loss: 0.032127
(Iteration 2761 / 5000) loss: 0.030468
(Iteration 2771 / 5000) loss: 0.037008
(Iteration 2781 / 5000) loss: 0.054076
(Iteration 2791 / 5000) loss: 0.034199
(Iteration 2801 / 5000) loss: 0.036159
(Iteration 2811 / 5000) loss: 0.030899
(Iteration 2821 / 5000) loss: 0.070932
(Iteration 2831 / 5000) loss: 0.024053
(Iteration 2841 / 5000) loss: 0.035270
(Iteration 2851 / 5000) loss: 0.025898
(Iteration 2861 / 5000) loss: 0.088055
(Iteration 2871 / 5000) loss: 0.058870
(Iteration 2881 / 5000) loss: 0.102871
(Iteration 2891 / 5000) loss: 0.020145
(Iteration 2901 / 5000) loss: 0.076126
(Iteration 2911 / 5000) loss: 0.084456
(Iteration 2921 / 5000) loss: 0.058761
(Iteration 2931 / 5000) loss: 0.040017
(Iteration 2941 / 5000) loss: 0.051068
(Iteration 2951 / 5000) loss: 0.042650
(Iteration 2961 / 5000) loss: 0.059668
(Iteration 2971 / 5000) loss: 0.039594
(Iteration 2981 / 5000) loss: 0.074025
(Iteration 2991 / 5000) loss: 0.059274
(Iteration 3001 / 5000) loss: 0.053715
(Iteration 3011 / 5000) loss: 0.033955
(Iteration 3021 / 5000) loss: 0.027305
(Iteration 3031 / 5000) loss: 0.036434
(Iteration 3041 / 5000) loss: 0.037711
(Iteration 3051 / 5000) loss: 0.024567
(Iteration 3061 / 5000) loss: 0.019330
(Iteration 3071 / 5000) loss: 0.056741
(Iteration 3081 / 5000) loss: 0.033976
(Iteration 3091 / 5000) loss: 0.037970
(Iteration 3101 / 5000) loss: 0.041788
(Iteration 3111 / 5000) loss: 0.041199
(Iteration 3121 / 5000) loss: 0.042894
(Iteration 3131 / 5000) loss: 0.045536
```

```
(Iteration 3141 / 5000) loss: 0.024420
(Iteration 3151 / 5000) loss: 0.070113
(Iteration 3161 / 5000) loss: 0.084537
(Iteration 3171 / 5000) loss: 0.039280
(Iteration 3181 / 5000) loss: 0.062656
(Iteration 3191 / 5000) loss: 0.031054
(Iteration 3201 / 5000) loss: 0.018972
(Iteration 3211 / 5000) loss: 0.066831
(Iteration 3221 / 5000) loss: 0.050220
(Iteration 3231 / 5000) loss: 0.040032
(Iteration 3241 / 5000) loss: 0.054585
(Iteration 3251 / 5000) loss: 0.039107
(Iteration 3261 / 5000) loss: 0.036374
(Iteration 3271 / 5000) loss: 0.048054
(Iteration 3281 / 5000) loss: 0.031696
(Iteration 3291 / 5000) loss: 0.074086
(Iteration 3301 / 5000) loss: 0.049917
(Iteration 3311 / 5000) loss: 0.052601
(Iteration 3321 / 5000) loss: 0.037278
(Iteration 3331 / 5000) loss: 0.028118
(Iteration 3341 / 5000) loss: 0.016693
(Iteration 3351 / 5000) loss: 0.058237
(Iteration 3361 / 5000) loss: 0.023223
(Iteration 3371 / 5000) loss: 0.034234
(Iteration 3381 / 5000) loss: 0.023802
(Iteration 3391 / 5000) loss: 0.024292
(Iteration 3401 / 5000) loss: 0.041935
(Iteration 3411 / 5000) loss: 0.028851
(Iteration 3421 / 5000) loss: 0.016171
(Iteration 3431 / 5000) loss: 0.035037
(Iteration 3441 / 5000) loss: 0.045557
(Iteration 3451 / 5000) loss: 0.026993
(Iteration 3461 / 5000) loss: 0.034992
(Iteration 3471 / 5000) loss: 0.034399
(Iteration 3481 / 5000) loss: 0.031287
(Iteration 3491 / 5000) loss: 0.049543
(Iteration 3501 / 5000) loss: 0.041938
(Iteration 3511 / 5000) loss: 0.026432
(Iteration 3521 / 5000) loss: 0.014573
(Iteration 3531 / 5000) loss: 0.043989
(Iteration 3541 / 5000) loss: 0.034629
(Iteration 3551 / 5000) loss: 0.047457
(Iteration 3561 / 5000) loss: 0.032869
(Iteration 3571 / 5000) loss: 0.030975
(Iteration 3581 / 5000) loss: 0.034374
(Iteration 3591 / 5000) loss: 0.029360
(Iteration 3601 / 5000) loss: 0.034517
(Iteration 3611 / 5000) loss: 0.032150
(Iteration 3621 / 5000) loss: 0.074807
(Iteration 3631 / 5000) loss: 0.045313
(Iteration 3641 / 5000) loss: 0.035010
(Iteration 3651 / 5000) loss: 0.047982
(Iteration 3661 / 5000) loss: 0.030952
(Iteration 3671 / 5000) loss: 0.021259
(Iteration 3681 / 5000) loss: 0.053226
(Iteration 3691 / 5000) loss: 0.064203
(Iteration 3701 / 5000) loss: 0.024083
(Iteration 3711 / 5000) loss: 0.021322
(Iteration 3721 / 5000) loss: 0.041338
(Iteration 3731 / 5000) loss: 0.023299
(Iteration 3741 / 5000) loss: 0.014141
(Iteration 3751 / 5000) loss: 0.028998
(Iteration 3761 / 5000) loss: 0.030180
(Iteration 3771 / 5000) loss: 0.024464
(Iteration 3781 / 5000) loss: 0.021665
(Iteration 3791 / 5000) loss: 0.029613
(Iteration 3801 / 5000) loss: 0.022322
(Iteration 3811 / 5000) loss: 0.024273
(Iteration 3821 / 5000) loss: 0.011466
(Iteration 3831 / 5000) loss: 0.039700
(Iteration 3841 / 5000) loss: 0.038757
(Iteration 3851 / 5000) loss: 0.037643
(Iteration 3861 / 5000) loss: 0.035864
(Iteration 3871 / 5000) loss: 0.023433
(Iteration 3881 / 5000) loss: 0.038423
(Iteration 3891 / 5000) loss: 0.015724
(Iteration 3901 / 5000) loss: 0.028129
(Iteration 3911 / 5000) loss: 0.026116
(Iteration 3921 / 5000) loss: 0.057014
(Iteration 3931 / 5000) loss: 0.043385
(Iteration 3941 / 5000) loss: 0.015868
(Iteration 3951 / 5000) loss: 0.021217
(Iteration 3961 / 5000) loss: 0.040107
(Iteration 3971 / 5000) loss: 0.027008
(Iteration 3981 / 5000) loss: 0.015068
(Iteration 3991 / 5000) loss: 0.021460
(Iteration 4001 / 5000) loss: 0.026698
(Iteration 4011 / 5000) loss: 0.023678
(Iteration 4021 / 5000) loss: 0.040953
(Iteration 4031 / 5000) loss: 0.044832
(Iteration 4041 / 5000) loss: 0.014249
(Iteration 4051 / 5000) loss: 0.019687
(Iteration 4061 / 5000) loss: 0.032964
(Iteration 4071 / 5000) loss: 0.029726
(Iteration 4081 / 5000) loss: 0.029178
(Iteration 4091 / 5000) loss: 0.033962
(Iteration 4101 / 5000) loss: 0.024902
(Iteration 4111 / 5000) loss: 0.010078
(Iteration 4121 / 5000) loss: 0.039600
(Iteration 4131 / 5000) loss: 0.015251
(Iteration 4141 / 5000) loss: 0.037625
(Iteration 4151 / 5000) loss: 0.026909
(Iteration 4161 / 5000) loss: 0.039265
(Iteration 4171 / 5000) loss: 0.067568
(Iteration 4181 / 5000) loss: 0.050671
```

```
(Iteration 4191 / 5000) loss: 0.026822
(Iteration 4201 / 5000) loss: 0.016860
(Iteration 4211 / 5000) loss: 0.013854
(Iteration 4221 / 5000) loss: 0.034788
(Iteration 4231 / 5000) loss: 0.027025
(Iteration 4241 / 5000) loss: 0.042791
(Iteration 4251 / 5000) loss: 0.023338
(Iteration 4261 / 5000) loss: 0.023062
(Iteration 4271 / 5000) loss: 0.021530
(Iteration 4281 / 5000) loss: 0.029499
(Iteration 4291 / 5000) loss: 0.014466
(Iteration 4301 / 5000) loss: 0.034534
(Iteration 4311 / 5000) loss: 0.020793
(Iteration 4321 / 5000) loss: 0.036590
(Iteration 4331 / 5000) loss: 0.021338
(Iteration 4341 / 5000) loss: 0.044449
(Iteration 4351 / 5000) loss: 0.021757
(Iteration 4361 / 5000) loss: 0.015433
(Iteration 4371 / 5000) loss: 0.035592
(Iteration 4381 / 5000) loss: 0.012812
(Iteration 4391 / 5000) loss: 0.020061
(Iteration 4401 / 5000) loss: 0.016843
(Iteration 4411 / 5000) loss: 0.047522
(Iteration 4421 / 5000) loss: 0.020902
(Iteration 4431 / 5000) loss: 0.021699
(Iteration 4441 / 5000) loss: 0.027268
(Iteration 4451 / 5000) loss: 0.036473
(Iteration 4461 / 5000) loss: 0.019258
(Iteration 4471 / 5000) loss: 0.032144
(Iteration 4481 / 5000) loss: 0.021610
(Iteration 4491 / 5000) loss: 0.040519
(Iteration 4501 / 5000) loss: 0.020248
(Iteration 4511 / 5000) loss: 0.024046
(Iteration 4521 / 5000) loss: 0.026689
(Iteration 4531 / 5000) loss: 0.035013
(Iteration 4541 / 5000) loss: 0.038092
(Iteration 4551 / 5000) loss: 0.021795
(Iteration 4561 / 5000) loss: 0.017306
(Iteration 4571 / 5000) loss: 0.024408
(Iteration 4581 / 5000) loss: 0.039591
(Iteration 4591 / 5000) loss: 0.038009
(Iteration 4601 / 5000) loss: 0.019653
(Iteration 4611 / 5000) loss: 0.030744
(Iteration 4621 / 5000) loss: 0.021717
(Iteration 4631 / 5000) loss: 0.035041
(Iteration 4641 / 5000) loss: 0.034605
(Iteration 4651 / 5000) loss: 0.030661
(Iteration 4661 / 5000) loss: 0.028397
(Iteration 4671 / 5000) loss: 0.023510
(Iteration 4681 / 5000) loss: 0.009300
(Iteration 4691 / 5000) loss: 0.016868
(Iteration 4701 / 5000) loss: 0.018807
(Iteration 4711 / 5000) loss: 0.038027
(Iteration 4721 / 5000) loss: 0.029547
(Iteration 4731 / 5000) loss: 0.027049
(Iteration 4741 / 5000) loss: 0.029920
(Iteration 4751 / 5000) loss: 0.019818
(Iteration 4761 / 5000) loss: 0.013648
(Iteration 4771 / 5000) loss: 0.034945
(Iteration 4781 / 5000) loss: 0.015210
(Iteration 4791 / 5000) loss: 0.016371
(Iteration 4801 / 5000) loss: 0.021306
(Iteration 4811 / 5000) loss: 0.018988
(Iteration 4821 / 5000) loss: 0.028656
(Iteration 4831 / 5000) loss: 0.011859
(Iteration 4841 / 5000) loss: 0.024290
(Iteration 4851 / 5000) loss: 0.026768
(Iteration 4861 / 5000) loss: 0.035850
(Iteration 4871 / 5000) loss: 0.031955
(Iteration 4881 / 5000) loss: 0.024955
(Iteration 4891 / 5000) loss: 0.027959
(Iteration 4901 / 5000) loss: 0.029178
(Iteration 4911 / 5000) loss: 0.030602
(Iteration 4921 / 5000) loss: 0.026819
(Iteration 4931 / 5000) loss: 0.019062
(Iteration 4941 / 5000) loss: 0.025788
(Iteration 4951 / 5000) loss: 0.034722
(Iteration 4961 / 5000) loss: 0.042984
(Iteration 4971 / 5000) loss: 0.014903
(Iteration 4981 / 5000) loss: 0.033723
(Iteration 4991 / 5000) loss: 0.014828
```



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 `cs231n/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.

Note: Some of the URLs are missing and will throw an error; re-run this cell until the output is at least 2 good caption samples.

In [17]:
```
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, urls):
        plt.imshow(image_from_url(url))
        plt.title('%s\n%s\nGT:%s' % (split, sample_caption, gt_caption))
        plt.axis('off')
        plt.show()
```

train

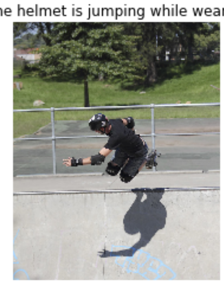
GT:<START> a woman is kneeling near some large <UNK> of food <END>

train

GT:<START> a group of men riding in a boat across a lake <END>

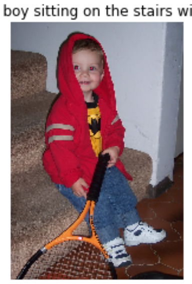
val

GT:<START> the man in the helmet is jumping while wearing <UNK> <UNK> <END>



val

GT:<START> a little boy sitting on the stairs with a racquet <END>



In [ ]: