CS7643: Deep Learning
Fall 2019
HW3 Solutions

Nicolas Six

October 23, 2019

1 Recurrent Neural Network

1.1 Vanilla RNN for parity function

Following the homework instructions, we are looking W and b such as:

$$[y_t] = act\left(W \cdot \begin{bmatrix} y_{t-1} \\ x_t \end{bmatrix} + b\right)$$

Following the instruction in the instructions, we can build the following truth table:

		y_{t-1}	
		0	1
x_t	0	0	1
	1	1	0

Table 1: Truth table of y_t

This table is basically the truth table of an XOR function. We saw during the last homework that such a function can't be represented by a single neurone and a linear activation.

Hopefully we are here free to use the activation function we want. If a linear activation is unable to represent XOR, an absolute activation function is a good fit.
Using the information from above we can quickly our neurone as:

$$[y_t] = abs\left(\begin{bmatrix} 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} y_{t-1} \\ x_t \end{bmatrix} + 0\right)$$

We can now apply this set of weight to the four possible incomes, as displayed on Table 2

If we then apply the activation function we chose to the value of Table 2, we easily fall back on the truth table of XOR, as showed on Table 3.
The above results assume that h is initialized with a value of zero.

		y_{t-1}	
		0	1
x_t	0	0	1
	1	-1	0

Table 2: Value table of our neurone for all possible inputs

		y_{t-1}	
		0	1
x_t	0	0	1
	1	1	0

Table 3: Value table output of our neurone for all possible inputs with the preset weights and bias as well as the selected activation function

1.2 LSTM for parity function

As said in Question 1.1, we want to build a network such as:

$$h_t = XOR\,(h_{t-1}, x_t)$$

In this question we have a lot of variable and a lot of different combination are so possible to get the correct results. We are here going to make it as simple as possible, to help the correction and the redaction of this report.

First we are going to set that $C_t = h_t$, this is easy to make by using the following weights:

$$W_o = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$
$$b_o = 1$$

Which give us $o_t = 1$, which translate in:

$$\begin{aligned} h_t &= o_t \cdot tanh(C_t) \\ &= tanh(C_t) \\ &= C_t \text{ for } C_t \in 0, 1 \end{aligned}$$

On the same idea we define $C_t^{\sim} = 1 - h_t$ with:

$$W_C = \begin{bmatrix} -1 \\ 0 \end{bmatrix}$$
$$b_C = 1$$

Now we want f_t to be 1 if x_t is zero and 0 otherwise:

$$W_t = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$
$$b_t = 1$$

Now we want i_t to be 1 if x_t is one and 0 otherwise:

$$W_t = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$
$$b_t = 0$$

As previously, we assume here that C_t and h_t are initialised to zero.

1.3 When to stop in beam search?

We know from the definition, that adding a new element in B_i have two different effect. First, as the final score is the product of probability, the score of the $b+1$ length sentences can only be smaller or equal than for the one of the highest scoring sentence of length b. Second, $best_{\leq i}$ can only be smaller or equal to $best_{\leq i+1}$ for obvious reason. It will stay the same if there is no sentence of length $i+1$ with higher score.

Which lead us the the conclusion that $best_{\leq i}$ is defined and is the highest scoring item in B_i. Then all new sentence in B_{i+1} would have a lower score than the highest scoring item in B_i, which here is $best_{\leq i}$, because of the first point cited previously. Which mean that any sentence longer than i would have lower or equal score than $best_{\leq i}$.

1.4 Exploding Gradients

For this question we will suppose that W is a diagonalizable matrix, which is implied by the fact that it's eigenvalues are given in the subject.
With that in mind, we know that it exist a matrix Q such as:

$$W^T = QDQ^{-1}$$

With D being a diagonal matrix with tha eigenvalues of W on the diagonal (and Q the eigenvectors).

Using this we can easily rewrite the given equation as:

$$\begin{aligned} h_t &= W^T h_{t-1} \\ &= QDQ^{-1} h_{t-1} \\ h_t &= QD^t Q^{-1} h_0 \end{aligned}$$

It is trivial that raising a diagonal matrix to a power t is equivalent to raise each of it's diagonal element to the power t. So depending on the value of the eigenvalues, raising the matrix D to a high power will make the diagonal element tend to zero (vanish) if they are smaller than one (in absolute value). If they are bigger than one (in absolute value) they will diverge to infinity. Or change signs / stay the same of they are 1 or -1.

This matrix is then used to compute the output, h_t, but also in the backward pass to propagate the gradient. Especially for the first inputs. So the gradient could be multiplied by values that are either too big (exploding) or too small (vanishing).

2 Coding: Sequence models for image captioning

2.1 RNN Captioning

RNN_Captioning

October 23, 2019

1 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_arra
        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))))
```

1.1 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 /home/nicolas/.local/lib/python3.6/site-packages (2.8.0)
Requirement already satisfied: six in /home/nicolas/.local/lib/python3.6/site-packages (from h5
Requirement already satisfied: numpy>=1.7 in /home/nicolas/.local/lib/python3.6/site-packages
```

2 Microsoft COCO

For this exercise we will use the 2014 release of the Microsoft COCO dataset 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]: # 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
```

2.1 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 [7]:
```
        # 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> and outside <UNK> area with a couch and umbrella <END>

<START> a pizza on a board has <UNK> tomatoes and <UNK> <END>

<START> a dog sits in a cage on the back of a motorcycle <END>

3 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`.

4 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 [8]:
```
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
```

5 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 [9]: 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))
```

```
dx error:  4.0192769090159184e-10
dprev_h error:  2.5632975303201374e-10
dWx error:  8.820222259148609e-10
dWh error:  4.703287554560559e-10
db error:  7.30162216654e-11
```

6 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 [10]:
```
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:  7.728466158305164e-08
```

7 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 [11]:
```
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('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:  1.5382468491701097e-09
dh0 error:  3.3839681556240896e-09
dWx error:  7.150535245339328e-09
dWh error:  1.297338408201546e-07
db error:  1.4889022954777414e-10
```

8 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 con-

vert words (represented by integers) into vectors. Run the following to check your implementation. You should see error around `1e-8`.

```
In [12]: 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
```

9 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 [13]: 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
```

10 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 [14]:
```
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:  2.9215945034030545e-10
dw error:  1.5772088618663602e-10
db error:  3.252200556967514e-11
```

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 [15]:
```
# 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, v

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

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

12 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 [16]: 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)
         expected_loss = 9.83235591003

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

```
loss:  9.832355910027387
expected loss:  9.83235591003
difference:  2.6130209107577684e-12
```

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

```
In [17]: 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=Fals
    e = rel_error(param_grad_num, grads[param_name])
    print('%s relative error: %e' % (param_name, e))
```

```
W_embed relative error: 2.331070e-09
W_proj relative error: 1.112417e-08
W_vocab relative error: 4.274379e-09
Wh relative error: 5.858117e-09
Wx relative error: 1.590657e-06
b relative error: 9.727211e-10
b_proj relative error: 1.934807e-08
b_vocab relative error: 7.087097e-11
```

13 Overfit small data

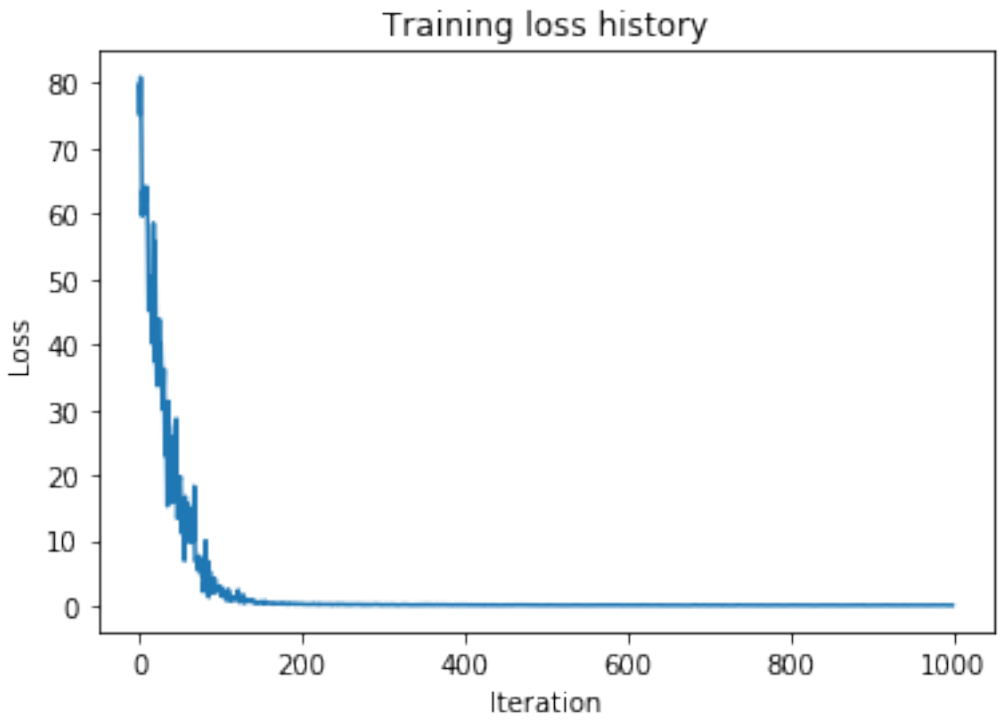
Similar to the `Solver` class that we used to train image classification models on the previous assignment, on this assignment we use a `CaptioningSolver` class to train image captioning models. Open the file `cs231n/captioning_solver.py` and read through the `CaptioningSolver` class; it should look 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 [22]:
```
np.random.seed(231)

small_data = load_coco_data(max_train=100)

small_rnn_model = CaptioningRNN(
          cell_type='rnn',
          word_to_idx=data['word_to_idx'],
          input_dim=data['train_features'].shape[1],
          hidden_dim=1024,
          wordvec_dim=256,
```

```
        )

    small_rnn_solver = CaptioningSolver(small_rnn_model, small_data,
               update_rule='adam',
               num_epochs=100,
               batch_size=10,
               optim_config={
                 'learning_rate': 1e-3,
               },
               lr_decay=0.95,
               verbose=True, print_every=100,
             )

    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 / 1000) loss: 79.682039
(Iteration 101 / 1000) loss: 2.252850
(Iteration 201 / 1000) loss: 0.332144
(Iteration 301 / 1000) loss: 0.178722
(Iteration 401 / 1000) loss: 0.165500
(Iteration 501 / 1000) loss: 0.149312
(Iteration 601 / 1000) loss: 0.156000
(Iteration 701 / 1000) loss: 0.109197
(Iteration 801 / 1000) loss: 0.118053
(Iteration 901 / 1000) loss: 0.083934
```



14 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 [24]: 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
a boy is jumping in the air on his skateboard <END>
GT:<START> a boy is jumping in the air on his skateboard <END>

train
a boy surfing a wave with a pink and yellow surf board <END>
GT:<START> a boy surfing a wave with a pink and yellow surf board <END>

val

a plate of food sitting on <END>

GT:<START> a man is taking a picture of some sandwiches <END>

val

woman cat sitting looking a on top of the dry <END>

GT:<START> doughnuts sitting on a brown bag on a table <END>

2.2 LSTM Captioning

LSTM_Captioning

October 23, 2019

1 Image Captioning with LSTMs

In the previous exercise you implemented a vanilla RNN and applied it to image captioning. In this notebook you will implement the LSTM update rule and use it for image captioning.

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
import nltk

from cs231n.gradient_check import eval_numerical_gradient, eval_numerical_gradient_arra
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))))
```

2 Load MS-COCO data

As in the previous notebook, we will use the Microsoft COCO dataset for captioning.

```
In [2]: # 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
```

3 LSTM

If you read recent papers, you'll see that many people use a variant on the vanialla RNN called Long-Short Term Memory (LSTM) RNNs. Vanilla RNNs can be tough to train on long sequences due to vanishing and exploding gradiants caused by repeated matrix multiplication. LSTMs solve this problem by replacing the simple update rule of the vanilla RNN with a gating mechanism as follows.

Similar to the vanilla RNN, at each timestep we receive an input $x_t \in \mathbb{R}^D$ and the previous hidden state $h_{t-1} \in \mathbb{R}^H$; the LSTM also maintains an H-dimensional *cell state*, so we also receive the previous cell state $c_{t-1} \in \mathbb{R}^H$. The learnable parameters of the LSTM are an *input-to-hidden* matrix $W_x \in \mathbb{R}^{4H \times D}$, a *hidden-to-hidden* matrix $W_h \in \mathbb{R}^{4H \times H}$ and a *bias vector* $b \in \mathbb{R}^{4H}$.

At each timestep we first compute an *activation vector* $a \in \mathbb{R}^{4H}$ as $a = W_x x_t + W_h h_{t-1} + b$. We then divide this into four vectors $a_i, a_f, a_o, a_g \in \mathbb{R}^H$ where a_i consists of the first H elements of a, a_f is the next H elements of a, etc. We then compute the *input gate* $g \in \mathbb{R}^H$, *forget gate* $f \in \mathbb{R}^H$, *output gate* $o \in \mathbb{R}^H$ and *block input* $g \in \mathbb{R}^H$ as

where σ is the sigmoid function and tanh is the hyperbolic tangent, both applied elementwise. Finally we compute the next cell state c_t and next hidden state h_t as

$$c_t = f \odot c_{t-1} + i \odot g \qquad\qquad h_t = o \odot \tanh(c_t)$$

where $\odot$ is the elementwise product of vectors.

In the rest of the notebook we will implement the LSTM update rule and apply it to the image captioning task.

In the code, we assume that data is stored in batches so that $X_t \in \mathbb{R}^{N\times D}$, and will work with *transposed* versions of the parameters: $W_x \in \mathbb{R}^{D\times 4H}$, $W_h \in \mathbb{R}^{H\times 4H}$ so that activations $A \in \mathbb{R}^{N\times 4H}$ can be computed efficiently as $A = X_t W_x + H_{t-1} W_h$

4 LSTM: step forward

Implement the forward pass for a single timestep of an LSTM in the `lstm_step_forward` function in the file `cs231n/rnn_layers.py`. This should be similar to the `rnn_step_forward` function that you implemented above, but using the LSTM update rule instead.

Once you are done, run the following to perform a simple test of your implementation. You should see errors around `1e-8` or less.

```
In [3]: N, D, H = 3, 4, 5
        x = np.linspace(-0.4, 1.2, num=N*D).reshape(N, D)
        prev_h = np.linspace(-0.3, 0.7, num=N*H).reshape(N, H)
        prev_c = np.linspace(-0.4, 0.9, num=N*H).reshape(N, H)
        Wx = np.linspace(-2.1, 1.3, num=4*D*H).reshape(D, 4 * H)
        Wh = np.linspace(-0.7, 2.2, num=4*H*H).reshape(H, 4 * H)
        b = np.linspace(0.3, 0.7, num=4*H)

        next_h, next_c, cache = lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)

        expected_next_h = np.asarray([
            [ 0.24635157,  0.28610883,  0.32240467,  0.35525807,  0.38474904],
            [ 0.49223563,  0.55611431,  0.61507696,  0.66844003,  0.7159181 ],
            [ 0.56735664,  0.66310127,  0.74419266,  0.80889665,  0.858299  ]])
        expected_next_c = np.asarray([
            [ 0.32986176,  0.39145139,  0.451556,    0.51014116,  0.56717407],
            [ 0.66382255,  0.76674007,  0.87195994,  0.97902709,  1.08751345],
            [ 0.74192008,  0.90592151,  1.07717006,  1.25120233,  1.42395676]])

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

```
next_h error:  5.7054131967097955e-09
next_c error:  5.8143123088804145e-09
```

5 LSTM: step backward

Implement the backward pass for a single LSTM timestep in the function `lstm_step_backward` in the file `cs231n/rnn_layers.py`. Once you are done, run the following to perform numeric gradient checking on your implementation. You should see errors around `1e-6` or less.

```
In [4]: np.random.seed(231)
```

```
N, D, H = 4, 5, 6
x = np.random.randn(N, D)
prev_h = np.random.randn(N, H)
prev_c = np.random.randn(N, H)
Wx = np.random.randn(D, 4 * H)
Wh = np.random.randn(H, 4 * H)
b = np.random.randn(4 * H)

next_h, next_c, cache = lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)

dnext_h = np.random.randn(*next_h.shape)
dnext_c = np.random.randn(*next_c.shape)

fx_h = lambda x: lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)[0]
fh_h = lambda h: lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)[0]
fc_h = lambda c: lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)[0]
fWx_h = lambda Wx: lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)[0]
fWh_h = lambda Wh: lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)[0]
fb_h = lambda b: lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)[0]

fx_c = lambda x: lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)[1]
fh_c = lambda h: lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)[1]
fc_c = lambda c: lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)[1]
fWx_c = lambda Wx: lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)[1]
fWh_c = lambda Wh: lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)[1]
fb_c = lambda b: lstm_step_forward(x, prev_h, prev_c, Wx, Wh, b)[1]

num_grad = eval_numerical_gradient_array

dx_num = num_grad(fx_h, x, dnext_h) + num_grad(fx_c, x, dnext_c)
dh_num = num_grad(fh_h, prev_h, dnext_h) + num_grad(fh_c, prev_h, dnext_c)
dc_num = num_grad(fc_h, prev_c, dnext_h) + num_grad(fc_c, prev_c, dnext_c)
dWx_num = num_grad(fWx_h, Wx, dnext_h) + num_grad(fWx_c, Wx, dnext_c)
dWh_num = num_grad(fWh_h, Wh, dnext_h) + num_grad(fWh_c, Wh, dnext_c)
db_num = num_grad(fb_h, b, dnext_h) + num_grad(fb_c, b, dnext_c)

dx, dh, dc, dWx, dWh, db = lstm_step_backward(dnext_h, dnext_c, cache)

print('dx error: ', rel_error(dx_num, dx))
print('dh error: ', rel_error(dh_num, dh))
print('dc error: ', rel_error(dc_num, dc))
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:  6.335163002532046e-10
dh error:  3.3963774090592634e-10
dc error:  1.5221723979041107e-10
```

```
dWx error:  2.1010960934639614e-09
dWh error:  9.712296109943072e-08
db error:  2.491522041931035e-10
```

6 LSTM: forward

In the function `lstm_forward` in the file `cs231n/rnn_layers.py`, implement the `lstm_forward` function to run an LSTM forward on an entire timeseries of data.

When you are done, run the following to check your implementation. You should see an error around `1e-7`.

```
In [5]: N, D, H, T = 2, 5, 4, 3
        x = np.linspace(-0.4, 0.6, num=N*T*D).reshape(N, T, D)
        h0 = np.linspace(-0.4, 0.8, num=N*H).reshape(N, H)
        Wx = np.linspace(-0.2, 0.9, num=4*D*H).reshape(D, 4 * H)
        Wh = np.linspace(-0.3, 0.6, num=4*H*H).reshape(H, 4 * H)
        b = np.linspace(0.2, 0.7, num=4*H)

        h, cache = lstm_forward(x, h0, Wx, Wh, b)

        expected_h = np.asarray([
         [[ 0.01764008,  0.01823233,  0.01882671,  0.0194232 ],
          [ 0.11287491,  0.12146228,  0.13018446,  0.13902939],
          [ 0.31358768,  0.33338627,  0.35304453,  0.37250975]],
         [[ 0.45767879,  0.4761092,   0.4936887,   0.51041945],
          [ 0.6704845,   0.69350089,  0.71486014,  0.7346449 ],
          [ 0.81733511,  0.83677871,  0.85403753,  0.86935314]]])

        print('h error: ', rel_error(expected_h, h))
```

```
h error:  8.610537452106624e-08
```

7 LSTM: backward

Implement the backward pass for an LSTM over an entire timeseries of data in the function `lstm_backward` in the file `cs231n/rnn_layers.py`. When you are done, run the following to perform numeric gradient checking on your implementation. You should see errors around `1e-7` or less.

```
In [6]: from cs231n.rnn_layers import lstm_forward, lstm_backward
        np.random.seed(231)

        N, D, T, H = 2, 3, 10, 6

        x = np.random.randn(N, T, D)
```

```
h0 = np.random.randn(N, H)
Wx = np.random.randn(D, 4 * H)
Wh = np.random.randn(H, 4 * H)
b = np.random.randn(4 * H)

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

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

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

fx = lambda x: lstm_forward(x, h0, Wx, Wh, b)[0]
fh0 = lambda h0: lstm_forward(x, h0, Wx, Wh, b)[0]
fWx = lambda Wx: lstm_forward(x, h0, Wx, Wh, b)[0]
fWh = lambda Wh: lstm_forward(x, h0, Wx, Wh, b)[0]
fb = lambda b: lstm_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:  6.9939005453315376e-09
dh0 error:  1.5042746972106784e-09
dWx error:  3.226295800444722e-09
dWh error:  2.6984653167426663e-06
db error:  8.23662763415198e-10
```

8 LSTM captioning model

Now that you have implemented an LSTM, update the implementation of the `loss` method of the `CaptioningRNN` class in the file `cs231n/classifiers/rnn.py` to handle the case where `self.cell_type` is `lstm`. This should require adding less than 10 lines of code.

Once you have done so, run the following to check your implementation. You should see a difference of less than `1e-10`.

In [7]:
```
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='lstm',
                  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(-0.5, 1.7, num=N*D).reshape(N, D)
        captions = (np.arange(N * T) % V).reshape(N, T)

        loss, grads = model.loss(features, captions)
        expected_loss = 9.82445935443

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

```
loss:  9.824459354432264
expected loss:  9.82445935443
difference:  2.2648549702353193e-12
```

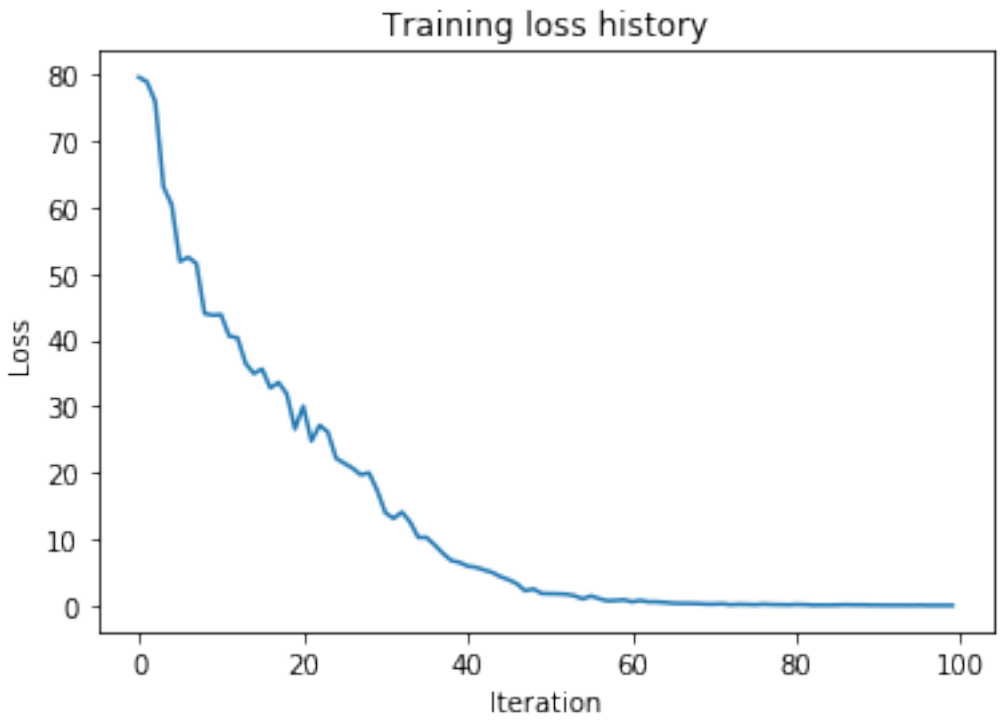
9 Overfit LSTM captioning model

Run the following to overfit an LSTM captioning model on the same small dataset as we used for the RNN previously. You should see losses less than 0.5.

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

        small_data = load_coco_data(max_train=50)

        small_lstm_model = CaptioningRNN(
                  cell_type='lstm',
                  word_to_idx=data['word_to_idx'],
                  input_dim=data['train_features'].shape[1],
                  hidden_dim=512,
                  wordvec_dim=256,
                  dtype=np.float32,
                )

        small_lstm_solver = CaptioningSolver(small_lstm_model, small_data,
                   update_rule='adam',
                   num_epochs=50,
```

```
                batch_size=25,
                optim_config={
                  'learning_rate': 5e-3,
                },
                lr_decay=0.995,
                verbose=True, print_every=10,
              )

        small_lstm_solver.train()

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

```
(Iteration 1 / 100) loss: 79.551150
(Iteration 11 / 100) loss: 43.829101
(Iteration 21 / 100) loss: 30.062616
(Iteration 31 / 100) loss: 14.020179
(Iteration 41 / 100) loss: 6.005075
(Iteration 51 / 100) loss: 1.849338
(Iteration 61 / 100) loss: 0.634570
(Iteration 71 / 100) loss: 0.285278
(Iteration 81 / 100) loss: 0.233098
(Iteration 91 / 100) loss: 0.121929
```



10 LSTM test-time sampling

Modify the `sample` method of the `CaptioningRNN` class to handle the case where `self.cell_type` is `lstm`. This should take fewer than 10 lines of code.

When you are done run the following to sample from your overfit LSTM model on some training and validation set samples.

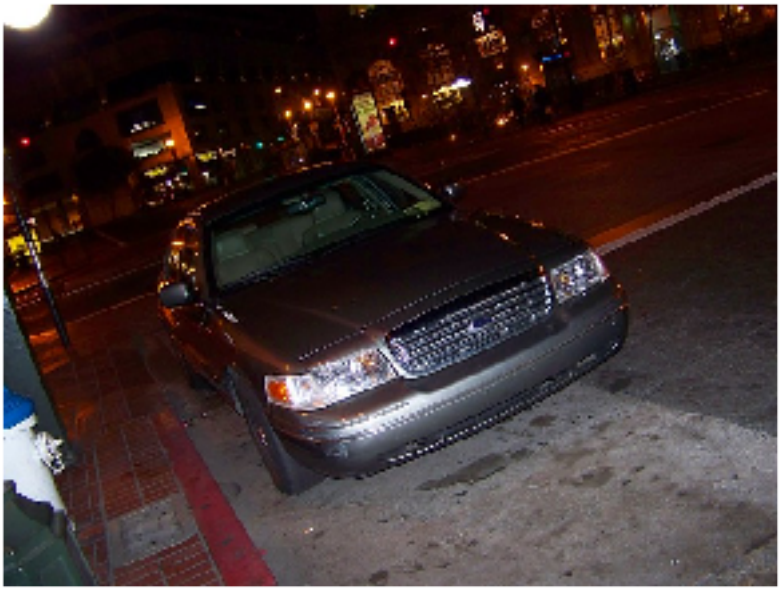
```
In [9]: for split in ['train', 'val']:
            minibatch = sample_coco_minibatch(small_data, split=split, batch_size=2)
            gt_captions, features, urls = minibatch
            gt_captions = decode_captions(gt_captions, data['idx_to_word'])

            sample_captions = small_lstm_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
a man standing on the side of a road with bags of luggage <END>
GT:<START> a man standing on the side of a road with bags of luggage <END>

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

val
a person <UNK> with a <UNK> <END>
GT:<START> a sign that is on the front of a train station <END>

val
a cat is <UNK> and a big <END>
GT:<START> a car is parked on a street at night <END>



11 Train a good captioning model (extra credit for 4803)

Using the pieces you have implemented in this and the previous notebook, train a captioning model that gives decent qualitative results (better than the random garbage you saw with the overfit models) when sampling on the validation set. You can subsample the training set if you want; we just want to see samples on the validation set that are better than random.

In addition to qualitatively evaluating your model by inspecting its results, you can also quantitatively evaluate your model using the BLEU unigram precision metric. In order to achieve full credit you should train a model that achieves a BLEU unigram score of >0.3. BLEU scores range from 0 to 1; the closer to 1, the better. Here's a reference to the paper that introduces BLEU if you're interested in learning more about how it works.

Feel free to use PyTorch for this section if you'd like to train faster on a GPU... though you can definitely get above 0.3 using your Numpy code. We're providing you the evaluation code that is compatible with the Numpy model as defined above... you should be able to adapt it for PyTorch if you go that route.

Create the model in the file `cs231n/classifiers/mymodel.py`. You can base it after the `CaptioningRNN` class. Write a text comment in the delineated cell below explaining what you tried in your model.

Also add a cell below that trains and tests your model. Make sure to include the call to evaluate_model which prints out your highest validation BLEU score for full credit.

```
In [10]: def BLEU_score(gt_caption, sample_caption):
             """
             gt_caption: string, ground-truth caption
             sample_caption: string, your model's predicted caption
             Returns unigram BLEU score.
             """
             reference = [x for x in gt_caption.split(' ')
                          if ('<END>' not in x and '<START>' not in x and '<UNK>' not in x)]
             hypothesis = [x for x in sample_caption.split(' ')
                           if ('<END>' not in x and '<START>' not in x and '<UNK>' not in x)]
             BLEUscore = nltk.translate.bleu_score.sentence_bleu([reference], hypothesis, weigh
             return BLEUscore

         def evaluate_model(model):
             """
             model: CaptioningRNN model
             Prints unigram BLEU score averaged over 1000 training and val examples.
             """
             BLEUscores = {}
             for split in ['train', 'val']:
                 minibatch = sample_coco_minibatch(data, split=split, batch_size=1000)
                 gt_captions, features, urls = minibatch
                 gt_captions = decode_captions(gt_captions, data['idx_to_word'])
```

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

            total_score = 0.0
            for gt_caption, sample_caption, url in zip(gt_captions, sample_captions, urls
                total_score += BLEU_score(gt_caption, sample_caption)

            BLEUscores[split] = total_score / len(sample_captions)

        for split in BLEUscores:
            print('Average BLEU score for %s: %f' % (split, BLEUscores[split]))
```

12 write a description of your model here:

Same model as the LSTM Numpy implementation, with fine tunned hyper parameters and more data. Training take times.

All paramters were changed, the most important one was probably the hidden dim (and the data quantity) a large batch size allowed to get the gradiant right while a lower learning rate was needed to take into account the large number of iteration vs the small number of epoch. This trainning was completed on around 12 hours on a 12 threads desktop Intel i7-8700 CPU.

In [14]:
```
# write your code to train your model here.
from cs231n.classifiers.mymodel import MyModel

my_data = load_coco_data(max_train=None)

my_lstm_model = MyModel(
          cell_type='lstm',
          word_to_idx=data['word_to_idx'],
          input_dim=data['train_features'].shape[1],
          hidden_dim=2048,
          wordvec_dim=256,
          dtype=np.float32,
        )

my_lstm_solver = CaptioningSolver(my_lstm_model, my_data,
           update_rule='adam',
           num_epochs=3,
           batch_size=1024,
           optim_config={
             'learning_rate': 1e-4,
           },
           lr_decay=0.9,
           verbose=True, print_every=1,
         )
```

```
        my_lstm_solver.train()

        # Plot the training losses
        plt.plot(my_lstm_solver.loss_history)
        plt.xlabel('Iteration')
        plt.ylabel('Loss')
        plt.title('Training loss history')
        plt.show()
        # make sure to include the call to evaluate_model which prints out your highest valid
        print("LSTM score:")
        evaluate_model(small_lstm_model)
        print("My model score:")
        evaluate_model(my_lstm_model)
```

```
(Iteration 1 / 1170) loss: 77.767695
(Iteration 2 / 1170) loss: 76.677468
(Iteration 3 / 1170) loss: 76.428032
(Iteration 4 / 1170) loss: 77.299044
(Iteration 5 / 1170) loss: 76.416580
(Iteration 6 / 1170) loss: 76.095531
(Iteration 7 / 1170) loss: 76.133898
(Iteration 8 / 1170) loss: 76.046990
(Iteration 9 / 1170) loss: 75.971192
(Iteration 10 / 1170) loss: 75.810697
(Iteration 11 / 1170) loss: 75.118970
(Iteration 12 / 1170) loss: 75.109374
(Iteration 13 / 1170) loss: 75.126804
(Iteration 14 / 1170) loss: 74.419421
(Iteration 15 / 1170) loss: 74.496195
(Iteration 16 / 1170) loss: 73.733022
(Iteration 17 / 1170) loss: 73.046076
(Iteration 18 / 1170) loss: 72.252211
(Iteration 19 / 1170) loss: 71.724380
(Iteration 20 / 1170) loss: 70.920663
(Iteration 21 / 1170) loss: 69.919237
(Iteration 22 / 1170) loss: 68.250768
(Iteration 23 / 1170) loss: 67.003846
(Iteration 24 / 1170) loss: 63.476444
(Iteration 25 / 1170) loss: 61.653228
(Iteration 26 / 1170) loss: 60.116551
(Iteration 27 / 1170) loss: 59.649217
(Iteration 28 / 1170) loss: 58.109839
(Iteration 29 / 1170) loss: 56.234807
(Iteration 30 / 1170) loss: 55.763547
(Iteration 31 / 1170) loss: 55.955834
(Iteration 32 / 1170) loss: 56.089155
(Iteration 33 / 1170) loss: 54.322441
(Iteration 34 / 1170) loss: 54.870358
```

```
(Iteration 35 / 1170) loss: 53.879896
(Iteration 36 / 1170) loss: 53.968145
(Iteration 37 / 1170) loss: 53.402411
(Iteration 38 / 1170) loss: 53.305978
(Iteration 39 / 1170) loss: 54.039428
(Iteration 40 / 1170) loss: 52.548754
(Iteration 41 / 1170) loss: 52.227963
(Iteration 42 / 1170) loss: 52.663542
(Iteration 43 / 1170) loss: 52.060988
(Iteration 44 / 1170) loss: 52.002039
(Iteration 45 / 1170) loss: 52.399127
(Iteration 46 / 1170) loss: 52.358968
(Iteration 47 / 1170) loss: 51.642286
(Iteration 48 / 1170) loss: 51.396217
(Iteration 49 / 1170) loss: 51.864611
(Iteration 50 / 1170) loss: 51.966383
(Iteration 51 / 1170) loss: 51.737526
(Iteration 52 / 1170) loss: 51.305973
(Iteration 53 / 1170) loss: 50.934631
(Iteration 54 / 1170) loss: 51.189282
(Iteration 55 / 1170) loss: 51.298938
(Iteration 56 / 1170) loss: 50.841360
(Iteration 57 / 1170) loss: 50.241599
(Iteration 58 / 1170) loss: 50.717055
(Iteration 59 / 1170) loss: 50.157340
(Iteration 60 / 1170) loss: 49.956023
(Iteration 61 / 1170) loss: 50.112098
(Iteration 62 / 1170) loss: 50.678542
(Iteration 63 / 1170) loss: 50.597938
(Iteration 64 / 1170) loss: 49.622576
(Iteration 65 / 1170) loss: 50.548298
(Iteration 66 / 1170) loss: 50.249122
(Iteration 67 / 1170) loss: 49.330679
(Iteration 68 / 1170) loss: 50.202429
(Iteration 69 / 1170) loss: 49.492862
(Iteration 70 / 1170) loss: 49.411362
(Iteration 71 / 1170) loss: 49.313297
(Iteration 72 / 1170) loss: 49.423218
(Iteration 73 / 1170) loss: 49.499765
(Iteration 74 / 1170) loss: 49.697251
(Iteration 75 / 1170) loss: 49.578988
(Iteration 76 / 1170) loss: 49.090266
(Iteration 77 / 1170) loss: 49.230603
(Iteration 78 / 1170) loss: 49.244343
(Iteration 79 / 1170) loss: 47.892459
(Iteration 80 / 1170) loss: 48.606674
(Iteration 81 / 1170) loss: 49.016801
(Iteration 82 / 1170) loss: 48.700789
```

```
(Iteration 83 / 1170) loss: 48.202232
(Iteration 84 / 1170) loss: 48.731822
(Iteration 85 / 1170) loss: 49.042803
(Iteration 86 / 1170) loss: 48.494692
(Iteration 87 / 1170) loss: 48.209117
(Iteration 88 / 1170) loss: 48.558433
(Iteration 89 / 1170) loss: 47.750019
(Iteration 90 / 1170) loss: 48.317716
(Iteration 91 / 1170) loss: 47.420152
(Iteration 92 / 1170) loss: 47.838794
(Iteration 93 / 1170) loss: 48.014534
(Iteration 94 / 1170) loss: 47.845847
(Iteration 95 / 1170) loss: 47.950040
(Iteration 96 / 1170) loss: 47.657809
(Iteration 97 / 1170) loss: 47.449246
(Iteration 98 / 1170) loss: 47.367459
(Iteration 99 / 1170) loss: 47.498363
(Iteration 100 / 1170) loss: 47.180294
(Iteration 101 / 1170) loss: 47.346829
(Iteration 102 / 1170) loss: 47.076618
(Iteration 103 / 1170) loss: 47.881994
(Iteration 104 / 1170) loss: 47.875658
(Iteration 105 / 1170) loss: 47.249143
(Iteration 106 / 1170) loss: 46.898673
(Iteration 107 / 1170) loss: 46.846252
(Iteration 108 / 1170) loss: 47.222841
(Iteration 109 / 1170) loss: 46.825828
(Iteration 110 / 1170) loss: 47.231530
(Iteration 111 / 1170) loss: 46.474325
(Iteration 112 / 1170) loss: 46.666806
(Iteration 113 / 1170) loss: 46.539856
(Iteration 114 / 1170) loss: 46.652779
(Iteration 115 / 1170) loss: 46.713771
(Iteration 116 / 1170) loss: 45.989613
(Iteration 117 / 1170) loss: 46.045492
(Iteration 118 / 1170) loss: 46.096047
(Iteration 119 / 1170) loss: 46.392055
(Iteration 120 / 1170) loss: 46.464379
(Iteration 121 / 1170) loss: 46.231250
(Iteration 122 / 1170) loss: 46.909585
(Iteration 123 / 1170) loss: 46.603247
(Iteration 124 / 1170) loss: 45.638339
(Iteration 125 / 1170) loss: 46.415875
(Iteration 126 / 1170) loss: 45.873616
(Iteration 127 / 1170) loss: 45.797680
(Iteration 128 / 1170) loss: 45.452052
(Iteration 129 / 1170) loss: 46.089094
(Iteration 130 / 1170) loss: 46.000802
```

```
(Iteration 131 / 1170) loss: 45.739796
(Iteration 132 / 1170) loss: 45.592924
(Iteration 133 / 1170) loss: 45.666668
(Iteration 134 / 1170) loss: 45.961203
(Iteration 135 / 1170) loss: 45.225444
(Iteration 136 / 1170) loss: 45.577216
(Iteration 137 / 1170) loss: 46.070733
(Iteration 138 / 1170) loss: 45.232288
(Iteration 139 / 1170) loss: 45.481815
(Iteration 140 / 1170) loss: 45.490336
(Iteration 141 / 1170) loss: 44.528148
(Iteration 142 / 1170) loss: 45.515740
(Iteration 143 / 1170) loss: 45.223131
(Iteration 144 / 1170) loss: 45.137258
(Iteration 145 / 1170) loss: 45.273124
(Iteration 146 / 1170) loss: 45.411724
(Iteration 147 / 1170) loss: 45.399061
(Iteration 148 / 1170) loss: 44.964269
(Iteration 149 / 1170) loss: 44.829047
(Iteration 150 / 1170) loss: 45.484237
(Iteration 151 / 1170) loss: 44.993933
(Iteration 152 / 1170) loss: 45.416461
(Iteration 153 / 1170) loss: 45.314940
(Iteration 154 / 1170) loss: 44.533855
(Iteration 155 / 1170) loss: 44.831100
(Iteration 156 / 1170) loss: 44.679098
(Iteration 157 / 1170) loss: 44.568478
(Iteration 158 / 1170) loss: 44.398188
(Iteration 159 / 1170) loss: 44.107840
(Iteration 160 / 1170) loss: 44.840325
(Iteration 161 / 1170) loss: 44.498880
(Iteration 162 / 1170) loss: 44.790704
(Iteration 163 / 1170) loss: 44.050921
(Iteration 164 / 1170) loss: 44.291695
(Iteration 165 / 1170) loss: 44.359331
(Iteration 166 / 1170) loss: 44.272403
(Iteration 167 / 1170) loss: 44.373596
(Iteration 168 / 1170) loss: 43.611820
(Iteration 169 / 1170) loss: 44.035699
(Iteration 170 / 1170) loss: 43.673011
(Iteration 171 / 1170) loss: 43.975509
(Iteration 172 / 1170) loss: 43.844133
(Iteration 173 / 1170) loss: 43.595980
(Iteration 174 / 1170) loss: 43.890164
(Iteration 175 / 1170) loss: 43.406976
(Iteration 176 / 1170) loss: 43.461014
(Iteration 177 / 1170) loss: 43.519062
(Iteration 178 / 1170) loss: 44.241603
```

```
(Iteration 179 / 1170) loss: 44.132432
(Iteration 180 / 1170) loss: 43.215449
(Iteration 181 / 1170) loss: 43.274510
(Iteration 182 / 1170) loss: 43.237087
(Iteration 183 / 1170) loss: 42.972738
(Iteration 184 / 1170) loss: 42.986238
(Iteration 185 / 1170) loss: 42.952254
(Iteration 186 / 1170) loss: 42.784279
(Iteration 187 / 1170) loss: 42.550178
(Iteration 188 / 1170) loss: 43.015374
(Iteration 189 / 1170) loss: 43.149192
(Iteration 190 / 1170) loss: 42.315573
(Iteration 191 / 1170) loss: 42.716880
(Iteration 192 / 1170) loss: 42.877381
(Iteration 193 / 1170) loss: 42.865653
(Iteration 194 / 1170) loss: 42.763375
(Iteration 195 / 1170) loss: 42.812318
(Iteration 196 / 1170) loss: 42.318806
(Iteration 197 / 1170) loss: 42.632422
(Iteration 198 / 1170) loss: 42.303834
(Iteration 199 / 1170) loss: 42.884653
(Iteration 200 / 1170) loss: 42.605453
(Iteration 201 / 1170) loss: 42.440193
(Iteration 202 / 1170) loss: 42.064387
(Iteration 203 / 1170) loss: 42.449570
(Iteration 204 / 1170) loss: 42.482660
(Iteration 205 / 1170) loss: 42.290552
(Iteration 206 / 1170) loss: 42.058097
(Iteration 207 / 1170) loss: 42.361119
(Iteration 208 / 1170) loss: 42.523963
(Iteration 209 / 1170) loss: 42.017710
(Iteration 210 / 1170) loss: 41.620603
(Iteration 211 / 1170) loss: 42.062492
(Iteration 212 / 1170) loss: 41.741603
(Iteration 213 / 1170) loss: 41.872650
(Iteration 214 / 1170) loss: 41.685994
(Iteration 215 / 1170) loss: 41.536312
(Iteration 216 / 1170) loss: 41.728559
(Iteration 217 / 1170) loss: 41.302924
(Iteration 218 / 1170) loss: 41.794809
(Iteration 219 / 1170) loss: 41.733729
(Iteration 220 / 1170) loss: 41.329921
(Iteration 221 / 1170) loss: 41.060036
(Iteration 222 / 1170) loss: 41.269787
(Iteration 223 / 1170) loss: 41.632154
(Iteration 224 / 1170) loss: 41.031976
(Iteration 225 / 1170) loss: 41.386163
(Iteration 226 / 1170) loss: 40.689704
```

```
(Iteration 227 / 1170) loss: 40.557224
(Iteration 228 / 1170) loss: 40.560539
(Iteration 229 / 1170) loss: 39.889544
(Iteration 230 / 1170) loss: 40.728646
(Iteration 231 / 1170) loss: 40.614973
(Iteration 232 / 1170) loss: 40.758140
(Iteration 233 / 1170) loss: 39.859161
(Iteration 234 / 1170) loss: 40.683163
(Iteration 235 / 1170) loss: 40.564665
(Iteration 236 / 1170) loss: 40.774434
(Iteration 237 / 1170) loss: 40.241752
(Iteration 238 / 1170) loss: 40.384921
(Iteration 239 / 1170) loss: 39.854520
(Iteration 240 / 1170) loss: 40.451350
(Iteration 241 / 1170) loss: 40.404257
(Iteration 242 / 1170) loss: 39.743010
(Iteration 243 / 1170) loss: 39.769595
(Iteration 244 / 1170) loss: 39.884279
(Iteration 245 / 1170) loss: 40.190491
(Iteration 246 / 1170) loss: 39.262348
(Iteration 247 / 1170) loss: 39.959583
(Iteration 248 / 1170) loss: 40.123362
(Iteration 249 / 1170) loss: 39.641548
(Iteration 250 / 1170) loss: 39.457753
(Iteration 251 / 1170) loss: 40.118132
(Iteration 252 / 1170) loss: 39.691670
(Iteration 253 / 1170) loss: 39.607326
(Iteration 254 / 1170) loss: 39.329432
(Iteration 255 / 1170) loss: 39.799508
(Iteration 256 / 1170) loss: 39.359292
(Iteration 257 / 1170) loss: 39.750022
(Iteration 258 / 1170) loss: 39.298881
(Iteration 259 / 1170) loss: 39.126028
(Iteration 260 / 1170) loss: 38.951631
(Iteration 261 / 1170) loss: 39.292853
(Iteration 262 / 1170) loss: 39.047585
(Iteration 263 / 1170) loss: 39.086530
(Iteration 264 / 1170) loss: 39.321965
(Iteration 265 / 1170) loss: 39.036885
(Iteration 266 / 1170) loss: 39.208651
(Iteration 267 / 1170) loss: 39.433960
(Iteration 268 / 1170) loss: 39.467407
(Iteration 269 / 1170) loss: 39.234491
(Iteration 270 / 1170) loss: 38.656227
(Iteration 271 / 1170) loss: 38.848158
(Iteration 272 / 1170) loss: 38.437916
(Iteration 273 / 1170) loss: 38.993820
(Iteration 274 / 1170) loss: 38.064838
```

```
(Iteration 275 / 1170) loss: 38.658895
(Iteration 276 / 1170) loss: 38.788289
(Iteration 277 / 1170) loss: 38.689332
(Iteration 278 / 1170) loss: 38.355277
(Iteration 279 / 1170) loss: 38.639645
(Iteration 280 / 1170) loss: 38.699589
(Iteration 281 / 1170) loss: 38.677545
(Iteration 282 / 1170) loss: 38.211806
(Iteration 283 / 1170) loss: 38.217266
(Iteration 284 / 1170) loss: 37.813418
(Iteration 285 / 1170) loss: 38.175288
(Iteration 286 / 1170) loss: 37.762331
(Iteration 287 / 1170) loss: 38.595443
(Iteration 288 / 1170) loss: 37.423078
(Iteration 289 / 1170) loss: 38.437761
(Iteration 290 / 1170) loss: 38.432665
(Iteration 291 / 1170) loss: 38.538297
(Iteration 292 / 1170) loss: 38.729402
(Iteration 293 / 1170) loss: 38.131254
(Iteration 294 / 1170) loss: 37.734463
(Iteration 295 / 1170) loss: 37.287980
(Iteration 296 / 1170) loss: 37.789491
(Iteration 297 / 1170) loss: 37.476460
(Iteration 298 / 1170) loss: 38.338936
(Iteration 299 / 1170) loss: 38.093569
(Iteration 300 / 1170) loss: 37.553555
(Iteration 301 / 1170) loss: 37.894343
(Iteration 302 / 1170) loss: 37.764545
(Iteration 303 / 1170) loss: 37.756380
(Iteration 304 / 1170) loss: 37.168449
(Iteration 305 / 1170) loss: 37.063910
(Iteration 306 / 1170) loss: 37.412661
(Iteration 307 / 1170) loss: 37.106485
(Iteration 308 / 1170) loss: 36.893730
(Iteration 309 / 1170) loss: 37.387682
(Iteration 310 / 1170) loss: 36.819233
(Iteration 311 / 1170) loss: 38.120049
(Iteration 312 / 1170) loss: 37.113423
(Iteration 313 / 1170) loss: 37.289421
(Iteration 314 / 1170) loss: 37.548930
(Iteration 315 / 1170) loss: 36.515248
(Iteration 316 / 1170) loss: 37.221050
(Iteration 317 / 1170) loss: 37.586564
(Iteration 318 / 1170) loss: 36.639467
(Iteration 319 / 1170) loss: 37.652570
(Iteration 320 / 1170) loss: 36.787975
(Iteration 321 / 1170) loss: 36.930902
(Iteration 322 / 1170) loss: 36.834535
```

```
(Iteration 323 / 1170) loss: 36.522763
(Iteration 324 / 1170) loss: 36.810106
(Iteration 325 / 1170) loss: 36.233186
(Iteration 326 / 1170) loss: 36.533646
(Iteration 327 / 1170) loss: 36.285106
(Iteration 328 / 1170) loss: 36.626752
(Iteration 329 / 1170) loss: 36.593596
(Iteration 330 / 1170) loss: 36.864118
(Iteration 331 / 1170) loss: 36.251899
(Iteration 332 / 1170) loss: 36.394771
(Iteration 333 / 1170) loss: 36.809553
(Iteration 334 / 1170) loss: 36.773089
(Iteration 335 / 1170) loss: 36.678765
(Iteration 336 / 1170) loss: 36.638502
(Iteration 337 / 1170) loss: 36.303901
(Iteration 338 / 1170) loss: 37.100036
(Iteration 339 / 1170) loss: 36.052165
(Iteration 340 / 1170) loss: 36.562661
(Iteration 341 / 1170) loss: 36.154390
(Iteration 342 / 1170) loss: 35.953621
(Iteration 343 / 1170) loss: 35.569254
(Iteration 344 / 1170) loss: 36.481109
(Iteration 345 / 1170) loss: 36.300008
(Iteration 346 / 1170) loss: 35.931193
(Iteration 347 / 1170) loss: 35.471512
(Iteration 348 / 1170) loss: 36.330368
(Iteration 349 / 1170) loss: 36.093303
(Iteration 350 / 1170) loss: 35.372241
(Iteration 351 / 1170) loss: 35.753220
(Iteration 352 / 1170) loss: 35.762376
(Iteration 353 / 1170) loss: 35.285290
(Iteration 354 / 1170) loss: 35.607083
(Iteration 355 / 1170) loss: 35.980406
(Iteration 356 / 1170) loss: 35.271952
(Iteration 357 / 1170) loss: 35.319230
(Iteration 358 / 1170) loss: 35.546687
(Iteration 359 / 1170) loss: 35.431842
(Iteration 360 / 1170) loss: 35.845402
(Iteration 361 / 1170) loss: 35.468820
(Iteration 362 / 1170) loss: 35.592040
(Iteration 363 / 1170) loss: 35.086945
(Iteration 364 / 1170) loss: 35.030865
(Iteration 365 / 1170) loss: 35.698697
(Iteration 366 / 1170) loss: 35.175815
(Iteration 367 / 1170) loss: 35.683949
(Iteration 368 / 1170) loss: 35.005929
(Iteration 369 / 1170) loss: 35.878771
(Iteration 370 / 1170) loss: 35.164890
```

```
(Iteration 371 / 1170) loss: 35.779726
(Iteration 372 / 1170) loss: 35.067489
(Iteration 373 / 1170) loss: 35.363157
(Iteration 374 / 1170) loss: 35.298500
(Iteration 375 / 1170) loss: 35.413370
(Iteration 376 / 1170) loss: 35.701537
(Iteration 377 / 1170) loss: 35.188464
(Iteration 378 / 1170) loss: 35.033024
(Iteration 379 / 1170) loss: 34.917874
(Iteration 380 / 1170) loss: 35.040486
(Iteration 381 / 1170) loss: 35.354873
(Iteration 382 / 1170) loss: 34.802797
(Iteration 383 / 1170) loss: 35.231405
(Iteration 384 / 1170) loss: 35.351056
(Iteration 385 / 1170) loss: 35.091705
(Iteration 386 / 1170) loss: 35.014706
(Iteration 387 / 1170) loss: 34.371536
(Iteration 388 / 1170) loss: 34.840043
(Iteration 389 / 1170) loss: 35.030096
(Iteration 390 / 1170) loss: 34.722944
(Iteration 391 / 1170) loss: 34.723152
(Iteration 392 / 1170) loss: 35.327981
(Iteration 393 / 1170) loss: 34.980425
(Iteration 394 / 1170) loss: 34.244882
(Iteration 395 / 1170) loss: 34.524224
(Iteration 396 / 1170) loss: 34.569559
(Iteration 397 / 1170) loss: 34.491346
(Iteration 398 / 1170) loss: 34.989186
(Iteration 399 / 1170) loss: 34.198076
(Iteration 400 / 1170) loss: 34.329287
(Iteration 401 / 1170) loss: 34.504971
(Iteration 402 / 1170) loss: 34.271596
(Iteration 403 / 1170) loss: 33.924323
(Iteration 404 / 1170) loss: 35.346391
(Iteration 405 / 1170) loss: 34.416545
(Iteration 406 / 1170) loss: 34.216970
(Iteration 407 / 1170) loss: 34.381268
(Iteration 408 / 1170) loss: 33.995623
(Iteration 409 / 1170) loss: 34.472861
(Iteration 410 / 1170) loss: 34.015123
(Iteration 411 / 1170) loss: 34.169643
(Iteration 412 / 1170) loss: 34.236644
(Iteration 413 / 1170) loss: 34.303773
(Iteration 414 / 1170) loss: 33.893645
(Iteration 415 / 1170) loss: 34.856960
(Iteration 416 / 1170) loss: 34.501651
(Iteration 417 / 1170) loss: 34.633713
(Iteration 418 / 1170) loss: 34.435334
```

```
(Iteration 419 / 1170) loss: 34.225337
(Iteration 420 / 1170) loss: 34.424701
(Iteration 421 / 1170) loss: 33.430618
(Iteration 422 / 1170) loss: 33.797776
(Iteration 423 / 1170) loss: 34.400342
(Iteration 424 / 1170) loss: 34.131678
(Iteration 425 / 1170) loss: 33.785200
(Iteration 426 / 1170) loss: 35.014840
(Iteration 427 / 1170) loss: 34.391265
(Iteration 428 / 1170) loss: 33.532482
(Iteration 429 / 1170) loss: 33.523491
(Iteration 430 / 1170) loss: 33.909504
(Iteration 431 / 1170) loss: 33.649231
(Iteration 432 / 1170) loss: 33.806889
(Iteration 433 / 1170) loss: 34.040699
(Iteration 434 / 1170) loss: 34.336868
(Iteration 435 / 1170) loss: 32.894914
(Iteration 436 / 1170) loss: 33.366539
(Iteration 437 / 1170) loss: 34.031822
(Iteration 438 / 1170) loss: 33.498356
(Iteration 439 / 1170) loss: 33.989025
(Iteration 440 / 1170) loss: 33.769363
(Iteration 441 / 1170) loss: 33.441488
(Iteration 442 / 1170) loss: 33.859746
(Iteration 443 / 1170) loss: 34.013168
(Iteration 444 / 1170) loss: 33.319034
(Iteration 445 / 1170) loss: 33.659680
(Iteration 446 / 1170) loss: 33.403771
(Iteration 447 / 1170) loss: 33.404118
(Iteration 448 / 1170) loss: 33.638741
(Iteration 449 / 1170) loss: 33.799630
(Iteration 450 / 1170) loss: 34.285606
(Iteration 451 / 1170) loss: 33.811048
(Iteration 452 / 1170) loss: 33.785121
(Iteration 453 / 1170) loss: 33.615105
(Iteration 454 / 1170) loss: 34.206907
(Iteration 455 / 1170) loss: 33.898393
(Iteration 456 / 1170) loss: 33.121223
(Iteration 457 / 1170) loss: 33.159011
(Iteration 458 / 1170) loss: 33.523095
(Iteration 459 / 1170) loss: 33.094843
(Iteration 460 / 1170) loss: 33.655474
(Iteration 461 / 1170) loss: 33.348471
(Iteration 462 / 1170) loss: 33.158638
(Iteration 463 / 1170) loss: 33.153592
(Iteration 464 / 1170) loss: 33.303603
(Iteration 465 / 1170) loss: 33.292811
(Iteration 466 / 1170) loss: 33.542595
```

```
(Iteration 467 / 1170) loss: 33.698990
(Iteration 468 / 1170) loss: 33.563057
(Iteration 469 / 1170) loss: 32.714531
(Iteration 470 / 1170) loss: 32.950740
(Iteration 471 / 1170) loss: 32.751261
(Iteration 472 / 1170) loss: 32.717498
(Iteration 473 / 1170) loss: 33.434564
(Iteration 474 / 1170) loss: 33.193851
(Iteration 475 / 1170) loss: 33.190127
(Iteration 476 / 1170) loss: 33.353590
(Iteration 477 / 1170) loss: 32.795355
(Iteration 478 / 1170) loss: 33.320396
(Iteration 479 / 1170) loss: 33.200352
(Iteration 480 / 1170) loss: 33.020380
(Iteration 481 / 1170) loss: 33.332993
(Iteration 482 / 1170) loss: 32.960230
(Iteration 483 / 1170) loss: 33.187581
(Iteration 484 / 1170) loss: 32.623760
(Iteration 485 / 1170) loss: 32.893347
(Iteration 486 / 1170) loss: 32.509182
(Iteration 487 / 1170) loss: 33.603365
(Iteration 488 / 1170) loss: 33.370716
(Iteration 489 / 1170) loss: 33.279617
(Iteration 490 / 1170) loss: 33.327044
(Iteration 491 / 1170) loss: 32.930262
(Iteration 492 / 1170) loss: 32.619970
(Iteration 493 / 1170) loss: 33.139073
(Iteration 494 / 1170) loss: 32.908674
(Iteration 495 / 1170) loss: 32.352402
(Iteration 496 / 1170) loss: 32.433681
(Iteration 497 / 1170) loss: 32.626423
(Iteration 498 / 1170) loss: 32.801581
(Iteration 499 / 1170) loss: 33.067539
(Iteration 500 / 1170) loss: 33.079359
(Iteration 501 / 1170) loss: 32.756391
(Iteration 502 / 1170) loss: 32.939887
(Iteration 503 / 1170) loss: 33.116890
(Iteration 504 / 1170) loss: 32.735933
(Iteration 505 / 1170) loss: 32.379567
(Iteration 506 / 1170) loss: 32.675950
(Iteration 507 / 1170) loss: 32.241672
(Iteration 508 / 1170) loss: 32.359516
(Iteration 509 / 1170) loss: 32.896862
(Iteration 510 / 1170) loss: 32.512261
(Iteration 511 / 1170) loss: 32.999227
(Iteration 512 / 1170) loss: 32.645263
(Iteration 513 / 1170) loss: 32.379482
(Iteration 514 / 1170) loss: 32.954073
```

```
(Iteration 515 / 1170) loss: 32.149876
(Iteration 516 / 1170) loss: 32.225248
(Iteration 517 / 1170) loss: 32.857355
(Iteration 518 / 1170) loss: 32.158181
(Iteration 519 / 1170) loss: 31.657619
(Iteration 520 / 1170) loss: 32.681091
(Iteration 521 / 1170) loss: 32.874352
(Iteration 522 / 1170) loss: 32.513895
(Iteration 523 / 1170) loss: 32.371625
(Iteration 524 / 1170) loss: 32.389979
(Iteration 525 / 1170) loss: 32.732699
(Iteration 526 / 1170) loss: 32.414199
(Iteration 527 / 1170) loss: 32.162697
(Iteration 528 / 1170) loss: 32.658137
(Iteration 529 / 1170) loss: 32.756132
(Iteration 530 / 1170) loss: 32.481291
(Iteration 531 / 1170) loss: 32.160260
(Iteration 532 / 1170) loss: 32.545895
(Iteration 533 / 1170) loss: 32.714621
(Iteration 534 / 1170) loss: 31.956267
(Iteration 535 / 1170) loss: 32.295389
(Iteration 536 / 1170) loss: 32.827657
(Iteration 537 / 1170) loss: 31.879670
(Iteration 538 / 1170) loss: 32.823324
(Iteration 539 / 1170) loss: 32.425885
(Iteration 540 / 1170) loss: 32.333622
(Iteration 541 / 1170) loss: 32.068654
(Iteration 542 / 1170) loss: 31.691115
(Iteration 543 / 1170) loss: 32.427284
(Iteration 544 / 1170) loss: 32.157501
(Iteration 545 / 1170) loss: 32.810479
(Iteration 546 / 1170) loss: 32.004972
(Iteration 547 / 1170) loss: 32.011385
(Iteration 548 / 1170) loss: 32.757092
(Iteration 549 / 1170) loss: 32.218461
(Iteration 550 / 1170) loss: 32.442353
(Iteration 551 / 1170) loss: 32.030827
(Iteration 552 / 1170) loss: 32.189922
(Iteration 553 / 1170) loss: 32.216837
(Iteration 554 / 1170) loss: 31.570212
(Iteration 555 / 1170) loss: 31.931412
(Iteration 556 / 1170) loss: 31.990476
(Iteration 557 / 1170) loss: 31.971198
(Iteration 558 / 1170) loss: 31.213170
(Iteration 559 / 1170) loss: 32.091260
(Iteration 560 / 1170) loss: 32.419030
(Iteration 561 / 1170) loss: 32.174844
(Iteration 562 / 1170) loss: 32.333550
```

```
(Iteration 563 / 1170) loss: 31.784384
(Iteration 564 / 1170) loss: 32.042295
(Iteration 565 / 1170) loss: 31.845305
(Iteration 566 / 1170) loss: 32.639774
(Iteration 567 / 1170) loss: 31.881697
(Iteration 568 / 1170) loss: 32.425990
(Iteration 569 / 1170) loss: 32.038801
(Iteration 570 / 1170) loss: 31.942124
(Iteration 571 / 1170) loss: 32.178334
(Iteration 572 / 1170) loss: 32.944392
(Iteration 573 / 1170) loss: 32.210219
(Iteration 574 / 1170) loss: 32.003747
(Iteration 575 / 1170) loss: 31.911046
(Iteration 576 / 1170) loss: 31.528676
(Iteration 577 / 1170) loss: 31.871120
(Iteration 578 / 1170) loss: 31.586415
(Iteration 579 / 1170) loss: 32.020349
(Iteration 580 / 1170) loss: 31.508032
(Iteration 581 / 1170) loss: 31.861422
(Iteration 582 / 1170) loss: 31.613971
(Iteration 583 / 1170) loss: 31.651657
(Iteration 584 / 1170) loss: 32.208650
(Iteration 585 / 1170) loss: 31.868022
(Iteration 586 / 1170) loss: 31.803422
(Iteration 587 / 1170) loss: 31.954733
(Iteration 588 / 1170) loss: 31.435778
(Iteration 589 / 1170) loss: 31.613750
(Iteration 590 / 1170) loss: 31.438791
(Iteration 591 / 1170) loss: 31.732276
(Iteration 592 / 1170) loss: 31.737930
(Iteration 593 / 1170) loss: 31.320335
(Iteration 594 / 1170) loss: 30.883567
(Iteration 595 / 1170) loss: 31.922990
(Iteration 596 / 1170) loss: 31.909592
(Iteration 597 / 1170) loss: 31.410951
(Iteration 598 / 1170) loss: 31.353476
(Iteration 599 / 1170) loss: 31.792170
(Iteration 600 / 1170) loss: 31.519190
(Iteration 601 / 1170) loss: 31.450883
(Iteration 602 / 1170) loss: 31.149214
(Iteration 603 / 1170) loss: 32.082070
(Iteration 604 / 1170) loss: 31.331303
(Iteration 605 / 1170) loss: 31.235649
(Iteration 606 / 1170) loss: 31.593371
(Iteration 607 / 1170) loss: 31.650304
(Iteration 608 / 1170) loss: 32.436133
(Iteration 609 / 1170) loss: 31.482398
(Iteration 610 / 1170) loss: 31.337651
```

```
(Iteration 611 / 1170) loss: 30.988248
(Iteration 612 / 1170) loss: 31.622641
(Iteration 613 / 1170) loss: 30.899109
(Iteration 614 / 1170) loss: 31.828537
(Iteration 615 / 1170) loss: 31.486409
(Iteration 616 / 1170) loss: 31.652059
(Iteration 617 / 1170) loss: 32.001517
(Iteration 618 / 1170) loss: 31.565428
(Iteration 619 / 1170) loss: 31.192610
(Iteration 620 / 1170) loss: 31.512387
(Iteration 621 / 1170) loss: 31.272529
(Iteration 622 / 1170) loss: 31.000339
(Iteration 623 / 1170) loss: 31.262548
(Iteration 624 / 1170) loss: 31.453422
(Iteration 625 / 1170) loss: 31.239857
(Iteration 626 / 1170) loss: 31.392993
(Iteration 627 / 1170) loss: 31.261834
(Iteration 628 / 1170) loss: 31.299854
(Iteration 629 / 1170) loss: 30.697519
(Iteration 630 / 1170) loss: 31.280023
(Iteration 631 / 1170) loss: 31.492544
(Iteration 632 / 1170) loss: 31.284192
(Iteration 633 / 1170) loss: 30.677539
(Iteration 634 / 1170) loss: 31.001599
(Iteration 635 / 1170) loss: 31.727386
(Iteration 636 / 1170) loss: 31.343166
(Iteration 637 / 1170) loss: 31.459740
(Iteration 638 / 1170) loss: 31.204257
(Iteration 639 / 1170) loss: 31.062858
(Iteration 640 / 1170) loss: 31.190932
(Iteration 641 / 1170) loss: 31.053907
(Iteration 642 / 1170) loss: 31.498789
(Iteration 643 / 1170) loss: 31.419102
(Iteration 644 / 1170) loss: 31.268347
(Iteration 645 / 1170) loss: 31.031574
(Iteration 646 / 1170) loss: 30.644795
(Iteration 647 / 1170) loss: 30.881615
(Iteration 648 / 1170) loss: 31.355442
(Iteration 649 / 1170) loss: 30.823586
(Iteration 650 / 1170) loss: 30.412247
(Iteration 651 / 1170) loss: 31.539554
(Iteration 652 / 1170) loss: 31.234986
(Iteration 653 / 1170) loss: 30.846187
(Iteration 654 / 1170) loss: 31.564078
(Iteration 655 / 1170) loss: 30.939418
(Iteration 656 / 1170) loss: 31.244746
(Iteration 657 / 1170) loss: 30.782312
(Iteration 658 / 1170) loss: 31.424211
```

```
(Iteration 659 / 1170) loss: 31.200938
(Iteration 660 / 1170) loss: 30.994386
(Iteration 661 / 1170) loss: 31.064688
(Iteration 662 / 1170) loss: 30.568168
(Iteration 663 / 1170) loss: 30.940868
(Iteration 664 / 1170) loss: 30.321513
(Iteration 665 / 1170) loss: 30.519844
(Iteration 666 / 1170) loss: 30.706499
(Iteration 667 / 1170) loss: 30.766145
(Iteration 668 / 1170) loss: 30.926961
(Iteration 669 / 1170) loss: 31.691334
(Iteration 670 / 1170) loss: 30.628253
(Iteration 671 / 1170) loss: 31.260639
(Iteration 672 / 1170) loss: 30.667350
(Iteration 673 / 1170) loss: 30.696374
(Iteration 674 / 1170) loss: 30.651947
(Iteration 675 / 1170) loss: 30.741168
(Iteration 676 / 1170) loss: 31.035634
(Iteration 677 / 1170) loss: 30.803979
(Iteration 678 / 1170) loss: 30.364515
(Iteration 679 / 1170) loss: 30.575737
(Iteration 680 / 1170) loss: 30.934694
(Iteration 681 / 1170) loss: 30.417181
(Iteration 682 / 1170) loss: 30.030339
(Iteration 683 / 1170) loss: 30.743342
(Iteration 684 / 1170) loss: 30.557240
(Iteration 685 / 1170) loss: 31.357963
(Iteration 686 / 1170) loss: 30.959012
(Iteration 687 / 1170) loss: 30.615136
(Iteration 688 / 1170) loss: 30.716380
(Iteration 689 / 1170) loss: 31.165203
(Iteration 690 / 1170) loss: 30.576011
(Iteration 691 / 1170) loss: 30.522063
(Iteration 692 / 1170) loss: 31.011963
(Iteration 693 / 1170) loss: 30.926742
(Iteration 694 / 1170) loss: 30.863776
(Iteration 695 / 1170) loss: 30.278503
(Iteration 696 / 1170) loss: 30.489433
(Iteration 697 / 1170) loss: 30.556171
(Iteration 698 / 1170) loss: 30.770118
(Iteration 699 / 1170) loss: 30.477635
(Iteration 700 / 1170) loss: 31.123422
(Iteration 701 / 1170) loss: 30.911183
(Iteration 702 / 1170) loss: 30.083101
(Iteration 703 / 1170) loss: 30.732318
(Iteration 704 / 1170) loss: 30.466698
(Iteration 705 / 1170) loss: 30.942748
(Iteration 706 / 1170) loss: 30.977583
```

```
(Iteration 707 / 1170) loss: 30.421405
(Iteration 708 / 1170) loss: 30.824954
(Iteration 709 / 1170) loss: 30.274673
(Iteration 710 / 1170) loss: 30.544743
(Iteration 711 / 1170) loss: 30.617080
(Iteration 712 / 1170) loss: 30.278116
(Iteration 713 / 1170) loss: 30.649610
(Iteration 714 / 1170) loss: 31.447936
(Iteration 715 / 1170) loss: 29.647310
(Iteration 716 / 1170) loss: 30.258734
(Iteration 717 / 1170) loss: 30.663590
(Iteration 718 / 1170) loss: 30.415042
(Iteration 719 / 1170) loss: 30.611797
(Iteration 720 / 1170) loss: 29.931483
(Iteration 721 / 1170) loss: 30.219494
(Iteration 722 / 1170) loss: 30.496907
(Iteration 723 / 1170) loss: 30.025258
(Iteration 724 / 1170) loss: 30.663286
(Iteration 725 / 1170) loss: 30.084793
(Iteration 726 / 1170) loss: 30.564744
(Iteration 727 / 1170) loss: 30.364411
(Iteration 728 / 1170) loss: 30.480105
(Iteration 729 / 1170) loss: 30.341136
(Iteration 730 / 1170) loss: 30.663227
(Iteration 731 / 1170) loss: 29.961526
(Iteration 732 / 1170) loss: 29.906971
(Iteration 733 / 1170) loss: 30.896059
(Iteration 734 / 1170) loss: 29.807159
(Iteration 735 / 1170) loss: 29.951755
(Iteration 736 / 1170) loss: 30.466407
(Iteration 737 / 1170) loss: 30.359518
(Iteration 738 / 1170) loss: 30.260981
(Iteration 739 / 1170) loss: 29.949665
(Iteration 740 / 1170) loss: 30.213810
(Iteration 741 / 1170) loss: 29.758538
(Iteration 742 / 1170) loss: 30.158706
(Iteration 743 / 1170) loss: 29.736850
(Iteration 744 / 1170) loss: 30.046070
(Iteration 745 / 1170) loss: 30.106753
(Iteration 746 / 1170) loss: 30.138610
(Iteration 747 / 1170) loss: 29.831538
(Iteration 748 / 1170) loss: 29.988514
(Iteration 749 / 1170) loss: 30.097480
(Iteration 750 / 1170) loss: 30.605264
(Iteration 751 / 1170) loss: 29.752435
(Iteration 752 / 1170) loss: 29.865327
(Iteration 753 / 1170) loss: 30.452962
(Iteration 754 / 1170) loss: 29.800002
```

```
(Iteration 755 / 1170) loss: 30.248824
(Iteration 756 / 1170) loss: 30.305423
(Iteration 757 / 1170) loss: 29.924702
(Iteration 758 / 1170) loss: 29.868316
(Iteration 759 / 1170) loss: 30.219769
(Iteration 760 / 1170) loss: 30.015015
(Iteration 761 / 1170) loss: 30.196301
(Iteration 762 / 1170) loss: 29.915171
(Iteration 763 / 1170) loss: 30.557069
(Iteration 764 / 1170) loss: 30.268717
(Iteration 765 / 1170) loss: 29.812550
(Iteration 766 / 1170) loss: 29.675822
(Iteration 767 / 1170) loss: 30.227092
(Iteration 768 / 1170) loss: 29.912520
(Iteration 769 / 1170) loss: 30.228331
(Iteration 770 / 1170) loss: 30.789146
(Iteration 771 / 1170) loss: 30.653373
(Iteration 772 / 1170) loss: 30.559330
(Iteration 773 / 1170) loss: 30.265227
(Iteration 774 / 1170) loss: 29.828299
(Iteration 775 / 1170) loss: 29.961820
(Iteration 776 / 1170) loss: 30.147577
(Iteration 777 / 1170) loss: 30.234534
(Iteration 778 / 1170) loss: 29.827498
(Iteration 779 / 1170) loss: 29.911936
(Iteration 780 / 1170) loss: 29.564321
(Iteration 781 / 1170) loss: 29.308052
(Iteration 782 / 1170) loss: 29.451483
(Iteration 783 / 1170) loss: 29.406844
(Iteration 784 / 1170) loss: 30.334439
(Iteration 785 / 1170) loss: 29.880549
(Iteration 786 / 1170) loss: 29.491546
(Iteration 787 / 1170) loss: 30.230358
(Iteration 788 / 1170) loss: 30.214262
(Iteration 789 / 1170) loss: 29.416769
(Iteration 790 / 1170) loss: 30.316025
(Iteration 791 / 1170) loss: 30.237704
(Iteration 792 / 1170) loss: 29.633130
(Iteration 793 / 1170) loss: 29.446157
(Iteration 794 / 1170) loss: 29.795023
(Iteration 795 / 1170) loss: 29.397883
(Iteration 796 / 1170) loss: 29.690523
(Iteration 797 / 1170) loss: 30.252153
(Iteration 798 / 1170) loss: 29.624090
(Iteration 799 / 1170) loss: 29.527680
(Iteration 800 / 1170) loss: 29.691524
(Iteration 801 / 1170) loss: 29.611999
(Iteration 802 / 1170) loss: 29.543875
```

```
(Iteration 803 / 1170) loss: 29.649962
(Iteration 804 / 1170) loss: 29.706596
(Iteration 805 / 1170) loss: 29.910750
(Iteration 806 / 1170) loss: 29.443388
(Iteration 807 / 1170) loss: 29.694111
(Iteration 808 / 1170) loss: 30.015025
(Iteration 809 / 1170) loss: 29.675088
(Iteration 810 / 1170) loss: 29.988073
(Iteration 811 / 1170) loss: 30.248558
(Iteration 812 / 1170) loss: 29.417537
(Iteration 813 / 1170) loss: 29.455726
(Iteration 814 / 1170) loss: 29.280775
(Iteration 815 / 1170) loss: 29.833859
(Iteration 816 / 1170) loss: 30.283126
(Iteration 817 / 1170) loss: 29.514039
(Iteration 818 / 1170) loss: 29.827126
(Iteration 819 / 1170) loss: 29.742891
(Iteration 820 / 1170) loss: 29.525329
(Iteration 821 / 1170) loss: 29.443408
(Iteration 822 / 1170) loss: 29.390431
(Iteration 823 / 1170) loss: 29.494032
(Iteration 824 / 1170) loss: 29.899679
(Iteration 825 / 1170) loss: 30.010498
(Iteration 826 / 1170) loss: 29.234286
(Iteration 827 / 1170) loss: 29.768504
(Iteration 828 / 1170) loss: 29.865409
(Iteration 829 / 1170) loss: 29.887463
(Iteration 830 / 1170) loss: 29.528616
(Iteration 831 / 1170) loss: 29.857553
(Iteration 832 / 1170) loss: 29.337965
(Iteration 833 / 1170) loss: 29.963302
(Iteration 834 / 1170) loss: 29.954629
(Iteration 835 / 1170) loss: 29.242946
(Iteration 836 / 1170) loss: 29.179522
(Iteration 837 / 1170) loss: 29.199969
(Iteration 838 / 1170) loss: 29.455871
(Iteration 839 / 1170) loss: 29.226501
(Iteration 840 / 1170) loss: 29.683579
(Iteration 841 / 1170) loss: 29.878047
(Iteration 842 / 1170) loss: 29.611193
(Iteration 843 / 1170) loss: 29.060409
(Iteration 844 / 1170) loss: 29.264774
(Iteration 845 / 1170) loss: 29.420155
(Iteration 846 / 1170) loss: 29.656248
(Iteration 847 / 1170) loss: 29.314509
(Iteration 848 / 1170) loss: 29.969110
(Iteration 849 / 1170) loss: 29.518102
(Iteration 850 / 1170) loss: 29.437379
```

```
(Iteration 851 / 1170) loss: 29.565912
(Iteration 852 / 1170) loss: 29.958328
(Iteration 853 / 1170) loss: 30.013861
(Iteration 854 / 1170) loss: 29.415234
(Iteration 855 / 1170) loss: 29.846257
(Iteration 856 / 1170) loss: 29.580604
(Iteration 857 / 1170) loss: 29.441412
(Iteration 858 / 1170) loss: 29.380164
(Iteration 859 / 1170) loss: 29.266742
(Iteration 860 / 1170) loss: 29.447438
(Iteration 861 / 1170) loss: 29.554470
(Iteration 862 / 1170) loss: 29.564196
(Iteration 863 / 1170) loss: 29.448294
(Iteration 864 / 1170) loss: 28.677227
(Iteration 865 / 1170) loss: 29.183660
(Iteration 866 / 1170) loss: 29.145472
(Iteration 867 / 1170) loss: 28.955083
(Iteration 868 / 1170) loss: 29.780463
(Iteration 869 / 1170) loss: 29.693645
(Iteration 870 / 1170) loss: 29.404003
(Iteration 871 / 1170) loss: 29.673254
(Iteration 872 / 1170) loss: 29.543564
(Iteration 873 / 1170) loss: 29.327617
(Iteration 874 / 1170) loss: 29.098608
(Iteration 875 / 1170) loss: 28.986232
(Iteration 876 / 1170) loss: 29.431353
(Iteration 877 / 1170) loss: 29.901224
(Iteration 878 / 1170) loss: 28.900096
(Iteration 879 / 1170) loss: 29.639258
(Iteration 880 / 1170) loss: 28.766523
(Iteration 881 / 1170) loss: 29.412814
(Iteration 882 / 1170) loss: 29.038512
(Iteration 883 / 1170) loss: 29.285687
(Iteration 884 / 1170) loss: 29.378319
(Iteration 885 / 1170) loss: 29.048335
(Iteration 886 / 1170) loss: 29.289341
(Iteration 887 / 1170) loss: 29.087414
(Iteration 888 / 1170) loss: 28.841154
(Iteration 889 / 1170) loss: 28.969185
(Iteration 890 / 1170) loss: 28.937992
(Iteration 891 / 1170) loss: 29.440500
(Iteration 892 / 1170) loss: 29.326471
(Iteration 893 / 1170) loss: 28.844422
(Iteration 894 / 1170) loss: 29.550295
(Iteration 895 / 1170) loss: 29.303578
(Iteration 896 / 1170) loss: 29.228292
(Iteration 897 / 1170) loss: 28.423250
(Iteration 898 / 1170) loss: 29.308155
```

```
(Iteration 899 / 1170) loss: 29.058048
(Iteration 900 / 1170) loss: 28.896022
(Iteration 901 / 1170) loss: 28.873905
(Iteration 902 / 1170) loss: 29.333270
(Iteration 903 / 1170) loss: 28.999651
(Iteration 904 / 1170) loss: 29.349183
(Iteration 905 / 1170) loss: 29.021035
(Iteration 906 / 1170) loss: 29.204844
(Iteration 907 / 1170) loss: 28.813579
(Iteration 908 / 1170) loss: 29.191886
(Iteration 909 / 1170) loss: 29.038770
(Iteration 910 / 1170) loss: 28.988775
(Iteration 911 / 1170) loss: 29.041550
(Iteration 912 / 1170) loss: 29.494369
(Iteration 913 / 1170) loss: 28.721753
(Iteration 914 / 1170) loss: 29.275927
(Iteration 915 / 1170) loss: 28.741681
(Iteration 916 / 1170) loss: 29.086756
(Iteration 917 / 1170) loss: 29.408208
(Iteration 918 / 1170) loss: 28.774981
(Iteration 919 / 1170) loss: 28.982913
(Iteration 920 / 1170) loss: 28.694968
(Iteration 921 / 1170) loss: 29.193428
(Iteration 922 / 1170) loss: 28.577763
(Iteration 923 / 1170) loss: 28.887023
(Iteration 924 / 1170) loss: 28.941964
(Iteration 925 / 1170) loss: 28.724439
(Iteration 926 / 1170) loss: 29.481298
(Iteration 927 / 1170) loss: 28.984663
(Iteration 928 / 1170) loss: 29.205435
(Iteration 929 / 1170) loss: 29.315587
(Iteration 930 / 1170) loss: 29.388203
(Iteration 931 / 1170) loss: 28.876854
(Iteration 932 / 1170) loss: 28.632212
(Iteration 933 / 1170) loss: 28.997329
(Iteration 934 / 1170) loss: 29.022588
(Iteration 935 / 1170) loss: 28.904659
(Iteration 936 / 1170) loss: 28.674420
(Iteration 937 / 1170) loss: 29.151163
(Iteration 938 / 1170) loss: 29.308904
(Iteration 939 / 1170) loss: 29.001241
(Iteration 940 / 1170) loss: 28.967791
(Iteration 941 / 1170) loss: 29.231370
(Iteration 942 / 1170) loss: 28.937086
(Iteration 943 / 1170) loss: 29.109665
(Iteration 944 / 1170) loss: 29.713266
(Iteration 945 / 1170) loss: 28.334617
(Iteration 946 / 1170) loss: 28.775995
```

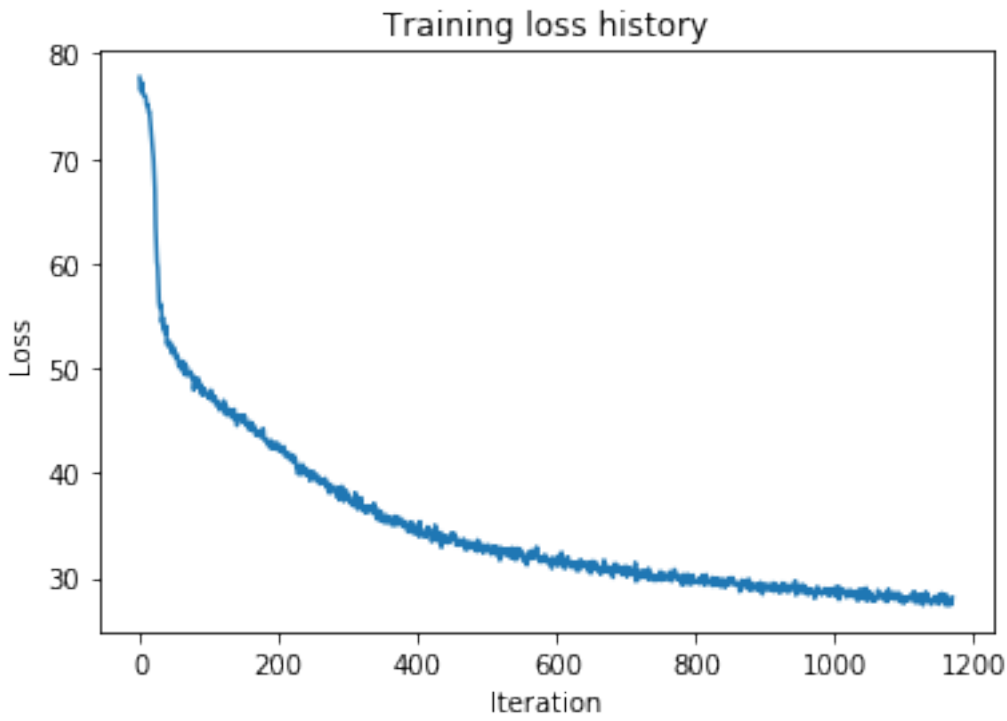
```
(Iteration 947 / 1170) loss: 29.232653
(Iteration 948 / 1170) loss: 29.115159
(Iteration 949 / 1170) loss: 28.616220
(Iteration 950 / 1170) loss: 28.965920
(Iteration 951 / 1170) loss: 28.701441
(Iteration 952 / 1170) loss: 28.798769
(Iteration 953 / 1170) loss: 29.322617
(Iteration 954 / 1170) loss: 29.350177
(Iteration 955 / 1170) loss: 29.022612
(Iteration 956 / 1170) loss: 28.597863
(Iteration 957 / 1170) loss: 29.242333
(Iteration 958 / 1170) loss: 28.732268
(Iteration 959 / 1170) loss: 29.137053
(Iteration 960 / 1170) loss: 28.787869
(Iteration 961 / 1170) loss: 28.745462
(Iteration 962 / 1170) loss: 28.367417
(Iteration 963 / 1170) loss: 28.904983
(Iteration 964 / 1170) loss: 28.719763
(Iteration 965 / 1170) loss: 28.908716
(Iteration 966 / 1170) loss: 28.532066
(Iteration 967 / 1170) loss: 28.935602
(Iteration 968 / 1170) loss: 28.511464
(Iteration 969 / 1170) loss: 28.452648
(Iteration 970 / 1170) loss: 28.664457
(Iteration 971 / 1170) loss: 28.713876
(Iteration 972 / 1170) loss: 28.034597
(Iteration 973 / 1170) loss: 28.469428
(Iteration 974 / 1170) loss: 28.909980
(Iteration 975 / 1170) loss: 28.846037
(Iteration 976 / 1170) loss: 29.155126
(Iteration 977 / 1170) loss: 28.632205
(Iteration 978 / 1170) loss: 28.676081
(Iteration 979 / 1170) loss: 28.913345
(Iteration 980 / 1170) loss: 28.578561
(Iteration 981 / 1170) loss: 28.613226
(Iteration 982 / 1170) loss: 28.369245
(Iteration 983 / 1170) loss: 28.622429
(Iteration 984 / 1170) loss: 28.725235
(Iteration 985 / 1170) loss: 28.684964
(Iteration 986 / 1170) loss: 28.844841
(Iteration 987 / 1170) loss: 28.972614
(Iteration 988 / 1170) loss: 28.634971
(Iteration 989 / 1170) loss: 28.618245
(Iteration 990 / 1170) loss: 28.777486
(Iteration 991 / 1170) loss: 28.716851
(Iteration 992 / 1170) loss: 28.281896
(Iteration 993 / 1170) loss: 28.949652
(Iteration 994 / 1170) loss: 28.427868
```

```
(Iteration 995 / 1170) loss: 28.289375
(Iteration 996 / 1170) loss: 28.440379
(Iteration 997 / 1170) loss: 28.737136
(Iteration 998 / 1170) loss: 28.970435
(Iteration 999 / 1170) loss: 28.585170
(Iteration 1000 / 1170) loss: 28.537002
(Iteration 1001 / 1170) loss: 28.853405
(Iteration 1002 / 1170) loss: 29.061061
(Iteration 1003 / 1170) loss: 28.239023
(Iteration 1004 / 1170) loss: 29.000617
(Iteration 1005 / 1170) loss: 29.135170
(Iteration 1006 / 1170) loss: 28.551154
(Iteration 1007 / 1170) loss: 27.920559
(Iteration 1008 / 1170) loss: 29.080961
(Iteration 1009 / 1170) loss: 28.593395
(Iteration 1010 / 1170) loss: 28.380071
(Iteration 1011 / 1170) loss: 28.559377
(Iteration 1012 / 1170) loss: 28.807022
(Iteration 1013 / 1170) loss: 28.904998
(Iteration 1014 / 1170) loss: 28.504866
(Iteration 1015 / 1170) loss: 28.719703
(Iteration 1016 / 1170) loss: 28.441040
(Iteration 1017 / 1170) loss: 28.676281
(Iteration 1018 / 1170) loss: 28.121957
(Iteration 1019 / 1170) loss: 28.628628
(Iteration 1020 / 1170) loss: 28.781378
(Iteration 1021 / 1170) loss: 28.425756
(Iteration 1022 / 1170) loss: 28.672018
(Iteration 1023 / 1170) loss: 28.333944
(Iteration 1024 / 1170) loss: 28.816378
(Iteration 1025 / 1170) loss: 28.234842
(Iteration 1026 / 1170) loss: 28.326254
(Iteration 1027 / 1170) loss: 28.500720
(Iteration 1028 / 1170) loss: 28.616619
(Iteration 1029 / 1170) loss: 28.139319
(Iteration 1030 / 1170) loss: 28.466767
(Iteration 1031 / 1170) loss: 28.304872
(Iteration 1032 / 1170) loss: 28.304780
(Iteration 1033 / 1170) loss: 28.742356
(Iteration 1034 / 1170) loss: 27.788925
(Iteration 1035 / 1170) loss: 28.178090
(Iteration 1036 / 1170) loss: 28.502421
(Iteration 1037 / 1170) loss: 28.184158
(Iteration 1038 / 1170) loss: 28.943661
(Iteration 1039 / 1170) loss: 28.355858
(Iteration 1040 / 1170) loss: 28.099197
(Iteration 1041 / 1170) loss: 28.421258
(Iteration 1042 / 1170) loss: 28.206057
```

```
(Iteration 1043 / 1170) loss: 28.677556
(Iteration 1044 / 1170) loss: 27.917520
(Iteration 1045 / 1170) loss: 28.776106
(Iteration 1046 / 1170) loss: 29.035018
(Iteration 1047 / 1170) loss: 28.320558
(Iteration 1048 / 1170) loss: 27.795305
(Iteration 1049 / 1170) loss: 28.242135
(Iteration 1050 / 1170) loss: 28.684234
(Iteration 1051 / 1170) loss: 28.017025
(Iteration 1052 / 1170) loss: 28.617441
(Iteration 1053 / 1170) loss: 27.915142
(Iteration 1054 / 1170) loss: 28.417297
(Iteration 1055 / 1170) loss: 28.184423
(Iteration 1056 / 1170) loss: 28.234795
(Iteration 1057 / 1170) loss: 27.771163
(Iteration 1058 / 1170) loss: 28.016220
(Iteration 1059 / 1170) loss: 27.763667
(Iteration 1060 / 1170) loss: 28.084024
(Iteration 1061 / 1170) loss: 28.222109
(Iteration 1062 / 1170) loss: 28.207939
(Iteration 1063 / 1170) loss: 28.810936
(Iteration 1064 / 1170) loss: 28.620083
(Iteration 1065 / 1170) loss: 28.137360
(Iteration 1066 / 1170) loss: 28.469947
(Iteration 1067 / 1170) loss: 28.240817
(Iteration 1068 / 1170) loss: 28.195230
(Iteration 1069 / 1170) loss: 28.395139
(Iteration 1070 / 1170) loss: 28.260399
(Iteration 1071 / 1170) loss: 27.894241
(Iteration 1072 / 1170) loss: 28.459563
(Iteration 1073 / 1170) loss: 28.466285
(Iteration 1074 / 1170) loss: 28.000565
(Iteration 1075 / 1170) loss: 28.528651
(Iteration 1076 / 1170) loss: 27.713833
(Iteration 1077 / 1170) loss: 27.870103
(Iteration 1078 / 1170) loss: 28.905975
(Iteration 1079 / 1170) loss: 28.506766
(Iteration 1080 / 1170) loss: 28.037633
(Iteration 1081 / 1170) loss: 28.502402
(Iteration 1082 / 1170) loss: 28.280396
(Iteration 1083 / 1170) loss: 28.022912
(Iteration 1084 / 1170) loss: 28.444705
(Iteration 1085 / 1170) loss: 28.305708
(Iteration 1086 / 1170) loss: 28.542622
(Iteration 1087 / 1170) loss: 28.522181
(Iteration 1088 / 1170) loss: 27.890638
(Iteration 1089 / 1170) loss: 28.381683
(Iteration 1090 / 1170) loss: 28.189118
```

```
(Iteration 1091 / 1170) loss: 28.368537
(Iteration 1092 / 1170) loss: 28.574107
(Iteration 1093 / 1170) loss: 28.481975
(Iteration 1094 / 1170) loss: 28.182890
(Iteration 1095 / 1170) loss: 27.715921
(Iteration 1096 / 1170) loss: 28.338725
(Iteration 1097 / 1170) loss: 28.129565
(Iteration 1098 / 1170) loss: 27.984473
(Iteration 1099 / 1170) loss: 27.612157
(Iteration 1100 / 1170) loss: 27.539009
(Iteration 1101 / 1170) loss: 27.875686
(Iteration 1102 / 1170) loss: 28.157067
(Iteration 1103 / 1170) loss: 27.916055
(Iteration 1104 / 1170) loss: 28.104660
(Iteration 1105 / 1170) loss: 27.860556
(Iteration 1106 / 1170) loss: 28.159342
(Iteration 1107 / 1170) loss: 27.953799
(Iteration 1108 / 1170) loss: 27.888603
(Iteration 1109 / 1170) loss: 28.002741
(Iteration 1110 / 1170) loss: 28.275730
(Iteration 1111 / 1170) loss: 28.347819
(Iteration 1112 / 1170) loss: 28.446450
(Iteration 1113 / 1170) loss: 28.089102
(Iteration 1114 / 1170) loss: 28.127514
(Iteration 1115 / 1170) loss: 28.686746
(Iteration 1116 / 1170) loss: 27.638460
(Iteration 1117 / 1170) loss: 28.304120
(Iteration 1118 / 1170) loss: 28.131632
(Iteration 1119 / 1170) loss: 28.362445
(Iteration 1120 / 1170) loss: 27.633282
(Iteration 1121 / 1170) loss: 28.073806
(Iteration 1122 / 1170) loss: 28.044797
(Iteration 1123 / 1170) loss: 28.375480
(Iteration 1124 / 1170) loss: 27.677307
(Iteration 1125 / 1170) loss: 27.814703
(Iteration 1126 / 1170) loss: 28.359944
(Iteration 1127 / 1170) loss: 28.001617
(Iteration 1128 / 1170) loss: 27.384863
(Iteration 1129 / 1170) loss: 28.181642
(Iteration 1130 / 1170) loss: 28.403436
(Iteration 1131 / 1170) loss: 28.191006
(Iteration 1132 / 1170) loss: 28.650015
(Iteration 1133 / 1170) loss: 28.089381
(Iteration 1134 / 1170) loss: 28.377556
(Iteration 1135 / 1170) loss: 27.724679
(Iteration 1136 / 1170) loss: 28.013137
(Iteration 1137 / 1170) loss: 27.693455
(Iteration 1138 / 1170) loss: 27.961595
```

```
(Iteration 1139 / 1170) loss: 27.907114
(Iteration 1140 / 1170) loss: 27.443694
(Iteration 1141 / 1170) loss: 27.968376
(Iteration 1142 / 1170) loss: 27.841702
(Iteration 1143 / 1170) loss: 27.846533
(Iteration 1144 / 1170) loss: 28.111039
(Iteration 1145 / 1170) loss: 27.837913
(Iteration 1146 / 1170) loss: 28.237898
(Iteration 1147 / 1170) loss: 27.659699
(Iteration 1148 / 1170) loss: 27.706018
(Iteration 1149 / 1170) loss: 28.220831
(Iteration 1150 / 1170) loss: 27.795767
(Iteration 1151 / 1170) loss: 27.472485
(Iteration 1152 / 1170) loss: 28.357459
(Iteration 1153 / 1170) loss: 27.775586
(Iteration 1154 / 1170) loss: 28.361578
(Iteration 1155 / 1170) loss: 28.505262
(Iteration 1156 / 1170) loss: 28.079944
(Iteration 1157 / 1170) loss: 27.727539
(Iteration 1158 / 1170) loss: 28.288190
(Iteration 1159 / 1170) loss: 27.729254
(Iteration 1160 / 1170) loss: 28.437921
(Iteration 1161 / 1170) loss: 27.352448
(Iteration 1162 / 1170) loss: 27.378884
(Iteration 1163 / 1170) loss: 28.022717
(Iteration 1164 / 1170) loss: 27.830514
(Iteration 1165 / 1170) loss: 27.779283
(Iteration 1166 / 1170) loss: 28.005814
(Iteration 1167 / 1170) loss: 27.377732
(Iteration 1168 / 1170) loss: 27.470774
(Iteration 1169 / 1170) loss: 27.504349
(Iteration 1170 / 1170) loss: 28.120129
```




```
LSTM score:
Average BLEU score for train: 0.175831
Average BLEU score for val: 0.168395
My model score:
Average BLEU score for train: 0.316566
Average BLEU score for val: 0.307564
```

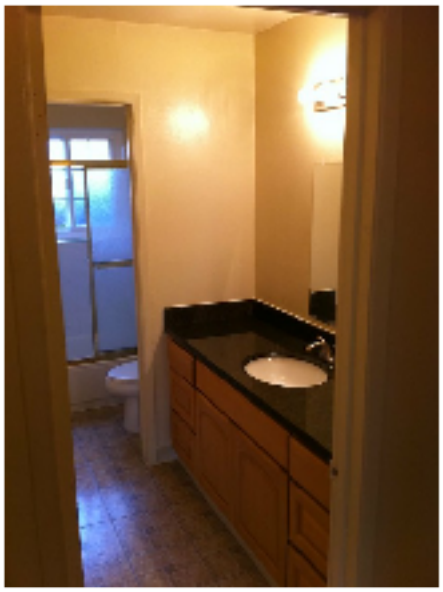
```
In [ ]: print("My model score:")
        evaluate_model(my_lstm_model)
```

```
In [17]: for split in ['train', 'val']:
             minibatch = sample_coco_minibatch(my_data, split=split, batch_size=2)
             gt_captions, features, urls = minibatch
             gt_captions = decode_captions(gt_captions, data['idx_to_word'])

             sample_captions = my_lstm_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
a dog is sitting on a bench with a <UNK> <END>
GT:<START> a <UNK> kitten walks in the grass around <UNK> foot <END>

train
a kitchen with a sink and a refrigerator <END>
GT:<START> a desk sitting next to a refrigerator microwave oven and mirror <END>

val
a bathroom with a toilet and a mirror in the mirror <END>
GT:<START> a small sink in a cabinet lies in a <UNK> <END>



val
a cat laying on top of a couch <END>
GT:<START> a cat sleeping on a bed with <UNK> <UNK> on top of him <END>

2.3 Transformer Classification

2.3.1 Ipython notebook

Transformer_Classification

October 23, 2019

1 Sentence Classification with Transformers

In this exercise you will implement a Transformer and use it to judge the grammaticality of English sentences.

A quick note: if you receive the following TypeError "super(type, obj): obj must be an instance or subtype of type", try restarting your kernel and re-running all cells. Once you have finished making changes to the model constuctor, you can avoid this issue by commenting out all of the model instantiations after the first (e.g. lines starting with "model = ClassificationTransformer").

```
In [1]: import numpy as np
        import csv
        import torch

        from gt_7643.transformer import ClassificationTransformer

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

1.1 The Corpus of Linguistic Acceptability (CoLA)

The Corpus of Linguistic Acceptability (CoLA) in its full form consists of 10657 sentences from 23 linguistics publications, expertly annotated for acceptability (grammaticality) by their original authors. Native English speakers consistently report a sharp contrast in acceptability between pairs of sentences. Some examples include:

`What did Betsy paint a picture of?` (Correct)

`What was a picture of painted by Betsy?` (Incorrect)

You can read more info about the dataset here. This is a binary classification task (predict 1 for correct grammar and 0 otherwise).

Can we train a neural network to accurately predict these human acceptability judgements? In this assignment, we will implement the forward pass of the Transformer architecture discussed in class. The general intuitive notion is that we will *encode* the sequence of tokens in the sentence, and then predict a binary output based on the final state that is the output of the model.

1.2 Load the preprocessed data

We've appended a "CLS" token to the beginning of each sequence, which can be used to make predictions. The benefit of appending this token to the beginning of the sequence (rather than the end) is that we can extract it quite easily (we don't need to remove paddings and figure out the length of each individual sequence in the batch). We'll come back to this.

We've additionally already constructed a vocabulary and converted all of the strings of tokens into integers which can be used for vocabulary lookup for you. Feel free to explore the data here.

```
In [2]: train_inxs = np.load('./gt_7643/datasets/train_inxs.npy')
        val_inxs = np.load('./gt_7643/datasets/val_inxs.npy')
        train_labels = np.load('./gt_7643/datasets/train_labels.npy')
        val_labels = np.load('./gt_7643/datasets/val_labels.npy')

        # load dictionary
        word_to_ix = {}
        with open("./gt_7643/datasets/word_to_ix.csv", "r") as f:
            reader = csv.reader(f)
            for line in reader:
                word_to_ix[line[0]] = line[1]
        print("Vocabulary Size:", len(word_to_ix))

        print(train_inxs.shape) # 7000 training instances, of (maximum/padded) length 43 words
        print(val_inxs.shape) # 1551 validation instances, of (maximum/padded) length 43 words
        print(train_labels.shape)
        print(val_labels.shape)

        # load checkers
        d1 = torch.load('./gt_7643/datasets/d1.pt')
        d2 = torch.load('./gt_7643/datasets/d2.pt')
        d3 = torch.load('./gt_7643/datasets/d3.pt')
        d4 = torch.load('./gt_7643/datasets/d4.pt')
```

```
Vocabulary Size: 1542
(7000, 43)
(1551, 43)
(7000,)
(1551,)
```

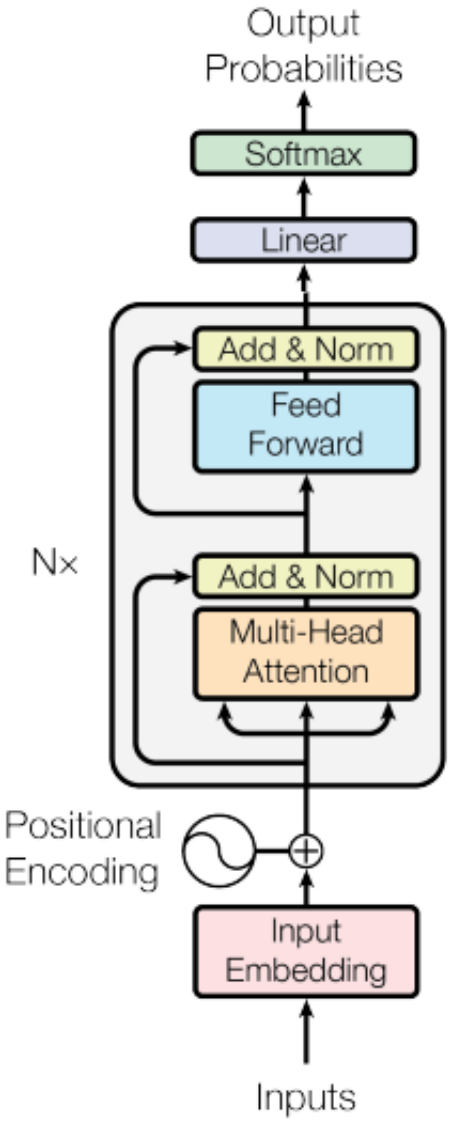
1.3 Transformers

We will be implementing a one-layer Transformer **encoder** which, similar to an RNN, can encode a sequence of inputs and produce a final output state for classification. This is the architecture:

You can refer to the original paper for more details.

Instead of using numpy for this model, we will be using Pytorch to implement the forward pass. You will not need to implement the backward pass for the various layers in this assigment.

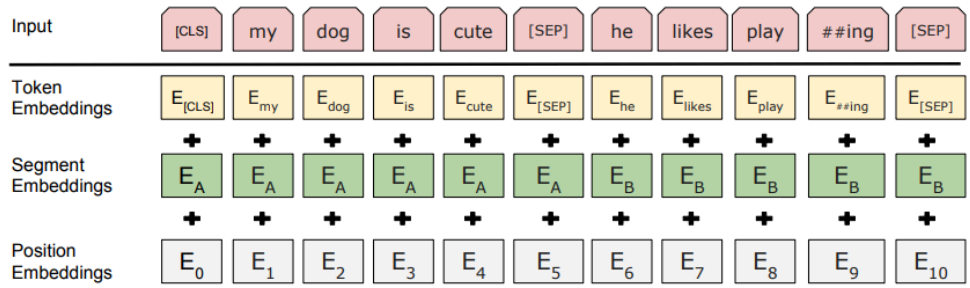
The file `gt_7643/transformer.py` contains the model class and methods for each layer. This is where you will write your implementations.



imgs/encoder.png

1.4 Deliverable 1: Embeddings

We will format our input embeddings similarly to how they are constructed in BERT (source of figure). Recall from lecture that unlike a RNN, a Transformer does not include any positional information about the order in which the words in the sentence occur. Because of this, we need to append a positional encoding token at each position. (We will ignore the segment embeddings and [SEP] token here, since we are only encoding one sentence at a time). We have already appended the [CLS] token for you in the previous step.



Your first task is to implement the embedding lookup, including the addition of positional encodings. Open the file `gt_7643/transformer.py` and complete all code parts for `Deliverable 1`.

```
In [3]: inputs = train_inxs[0:2]
        inputs = torch.LongTensor(inputs)

        model = ClassificationTransformer(word_to_ix, hidden_dim=128, num_heads=2, dim_feedfor
                                          dim_v=96, dim_q=96, max_length=train_inxs.shape[1])

        embeds = model.embed(inputs)

        try:
            print("Difference:", torch.sum(torch.pairwise_distance(embeds, d1)).item()) # shou
        except:
            print("NOT IMPLEMENTED")

Difference: 0.0017998494440689683
```

1.5 Deliverable 2: Multi-head Self-Attention

Attention can be computed in matrix-form using the following formula:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

We want to have multiple self-attention operations, computed in parallel. Each of these is called a *head*. We concatenate the heads and multiply them with the matrix `attention_head_projection` to produce the output of this layer.

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

imgs/ffn.png

After every multi-head self-attention and feedforward layer, there is a residual connection + layer normalization. Make sure to implement this, using the following formula:

$$\text{LayerNorm}(x + \text{Sublayer}(x))$$

Open the file `gt_7643/transformer.py` and implement the `multihead_attention` function. We have already initialized all of the layers you will need in the constructor.

```
In [4]: hidden_states = model.multi_head_attention(embeds)

        try:
            print("Difference:", torch.sum(torch.pairwise_distance(hidden_states, d2)).item())
        except:
            print("NOT IMPLEMENTED")
```

```
Difference: 0.001714545302093029
```

1.6 Deliverable 3: Element-Wise Feed-forward Layer

Open the file `gt_7643/transformer.py` and complete code for `Deliverable 3`: the element-wise feed-forward layer consisting of two linear transformers with a ReLU layer in between.

```
In [5]: # model = ClassificationTransformer(word_to_ix, hidden_dim=128, num_heads=2, dim_feedf
        #                                   dim_v=96, dim_q=96, max_length=train_inxs.shape[1]

        outputs = model.feedforward_layer(hidden_states)

        try:
            print("Difference:", torch.sum(torch.pairwise_distance(outputs, d3)).item()) # sho
        except:
            print("NOT IMPLEMENTED")
```

```
Difference: 0.0017153965309262276
```

1.7 Deliverable 4: Final Layer

Open the file `gt_7643/transformer.py` and complete code for `Deliverable 4`, to produce binary classification scores for the inputs based on the output of the Transformer.

```
In [6]: # model = ClassificationTransformer(word_to_ix, hidden_dim=128, num_heads=2, dim_feedf
        #                                   dim_v=96, dim_q=96, max_length=train_inxs.shape[1]
```

```
        scores = model.final_layer(outputs)

        try:
            print("Difference:", torch.sum(torch.pairwise_distance(scores, d4)).item()) # shou
        except Exception as e:
            print(e)
            print("NOT IMPLEMENTED")
```

```
Difference: 0.432945191860199
```

1.8 Deliverable 5: Putting it all together

Open the file `gt_7643/transformer.py` and complete the method `forward`, by putting together all of the methods you have developed in the right order to perform a full forward pass.

```
In [7]: inputs = train_inxs[0:2]
        inputs = torch.LongTensor(inputs)

        outputs = model.forward(inputs)

        try:
            print("Difference:", torch.sum(torch.pairwise_distance(outputs, scores)).item()) #
        except:
            print("NOT IMPLEMENTED")
```

```
Difference: 1.9999999949504854e-06
```

Great! We've just implemented a Transformer forward pass for text classification. One of the big perks of using PyTorch is that with a simple training loop, we can rely on automatic differentation (autograd) to do the work of the backward pass for us. This is not required for this assignment, but you can explore this on your own.

Make sure when you submit your PDF for this assignment to also include a copy of `transformer.py` converted to PDF as well.

2.3.2 Python code

```
# Code by Sarah Wiegreffe (saw@gatech.edu)
# Fall 2019

import numpy as np

import torch
from torch import nn
import random


# Do not modify these imports.

class ClassificationTransformer(nn.Module):
    """
    A single-layer Transformer which encodes a sequence of text and
    performs binary classification.

    The model has a vocab size of V, works on
    sequences of length T, has an hidden dimension of H, uses word vectors
    also of dimension H, and operates on minibatches of size N.
    """

    def __init__(self, word_to_ix, hidden_dim=128, num_heads=2, dim_feedforward=2048, dim_k
    =96, dim_v=96, dim_q=96,
                 max_length=43):
        '''
        :param word_to_ix: dictionary mapping words to unique indices
        :param hidden_dim: the dimensionality of the output embeddings that go into the
    final layer
        :param num_heads: the number of Transformer heads to use
        :param dim_feedforward: the dimension of the feedforward network model
        :param dim_k: the dimensionality of the key vectors
        :param dim_q: the dimensionality of the query vectors
        :param dim_v: the dimensionality of the value vectors
        '''
        super(ClassificationTransformer, self).__init__()
        assert hidden_dim % num_heads == 0

        self.num_heads = num_heads
        self.word_embedding_dim = hidden_dim
        self.hidden_dim = hidden_dim
        self.dim_feedforward = dim_feedforward
        self.max_length = max_length
        self.vocab_size = len(word_to_ix)

        self.dim_k = dim_k
        self.dim_v = dim_v
        self.dim_q = dim_q

        seed_torch(0)

        ##############################################################################
        # Deliverable 1: Initialize what you need for the embedding lookup (1 line). #
        # Hint: you will need to use the max_length parameter above.                 #
        ##############################################################################
        self.embedding_voc = nn.Embedding(num_embeddings=self.vocab_size, embedding_dim=self
    .hidden_dim)
        self.embedding_pos = nn.Embedding(num_embeddings=self.max_length, embedding_dim=self
    .hidden_dim)
        self.sqrt_dim_k = torch.sqrt(torch.Tensor([self.dim_k]))
        ##############################################################################
        #                                END OF YOUR CODE                            #
        ##############################################################################

        ##############################################################################
        # Deliverable 2: Initializations for multi-head self-attention.              #
```

```
        # You don't need to do anything here. Do not modify this code.                #
        ##############################################################################

        # Head #1
        self.k1 = nn.Linear(self.hidden_dim, self.dim_k)
        self.v1 = nn.Linear(self.hidden_dim, self.dim_v)
        self.q1 = nn.Linear(self.hidden_dim, self.dim_q)

        # Head #2
        self.k2 = nn.Linear(self.hidden_dim, self.dim_k)
        self.v2 = nn.Linear(self.hidden_dim, self.dim_v)
        self.q2 = nn.Linear(self.hidden_dim, self.dim_q)

        self.softmax = nn.Softmax(dim=2)
        self.attention_head_projection = nn.Linear(self.dim_v * self.num_heads, self.
    hidden_dim)
        self.norm_mh = nn.LayerNorm(self.hidden_dim)

        ##############################################################################
        # Deliverable 3: Initialize what you need for the feed-forward layer.        #
        # Don't forget the layer normalization.                                      #
        ##############################################################################
        self.relu = nn.ReLU()
        self.ff_transform1 = nn.Linear(self.hidden_dim, self.dim_feedforward)
        self.ff_transform2 = nn.Linear(self.dim_feedforward, self.hidden_dim)
        self.norm_ff = nn.LayerNorm(self.hidden_dim)
        ##############################################################################
        #                                END OF YOUR CODE                            #
        ##############################################################################

        ##############################################################################
        # Deliverable 4: Initialize what you need for the final layer (1-2 lines).   #
        ##############################################################################
        self.sigmoid = nn.Sigmoid()
        self.output_layer = nn.Linear(self.hidden_dim * self.max_length, 1)
        ##############################################################################
        #                                END OF YOUR CODE                            #
        ##############################################################################

    def forward(self, inputs):
        '''
        This function computes the full Transformer forward pass.
        Put together all of the layers you've developed in the correct order.

        :param inputs: a PyTorch tensor of shape (N,T). These are integer lookups.

        :returns: the model outputs. Should be normalized scores of shape (N,1).
        '''
        # outputs = None
        ##############################################################################
        # Deliverable 5: Implement the full Transformer stack for the forward pass. #
        # You will need to use all of the methods you have previously defined above.#
        # You should only be calling ClassificationTransformer class methods here.  #
        ##############################################################################
        outputs = self.final_layer(self.feedforward_layer(self.multi_head_attention(self.
    embed(inputs))))
        ##############################################################################
        #                                END OF YOUR CODE                            #
        ##############################################################################
        return outputs

    def embed(self, inputs):
        """
        :param inputs: intTensor of shape (N,T)
        :returns embeddings: floatTensor of shape (N,T,H)
        """
        # embeddings = None
        ##############################################################################
```

```
        # Deliverable 1: Implement the embedding lookup.                            #
        # Note: word_to_ix has keys from 0 to self.vocab_size - 1                   #
        # This will take a few lines.                                               #
        #############################################################################
        embeddings = self.embedding_voc(inputs)
        N, T = inputs.shape
        r = torch.arange(T).reshape((1, T))
        c = torch.ones((N, 1), dtype=torch.long)
        p = torch.mm(c, r)
        embeddings += self.embedding_pos(p)
        #############################################################################
        #                               END OF YOUR CODE                            #
        #############################################################################
        return embeddings

    def multi_head_attention(self, inputs):
        """
        :param inputs: float32 Tensor of shape (N,T,H)
        :returns outputs: float32 Tensor of shape (N,T,H)

        Traditionally we'd include a padding mask here, so that pads are ignored.
        This is a simplified implementation.
        """
        # outputs = None
        #############################################################################
        # Deliverable 2: Implement multi-head self-attention followed by add + norm.#
        # Use the provided 'Deliverable 2' layers initialized in the constructor.   #
        #############################################################################
        head1 = self.softmax(self.q1(inputs) @ self.k1(inputs).transpose(1, 2) / self.
    sqrt_dim_k) @ self.v1(inputs)
        head2 = self.softmax(self.q2(inputs) @ self.k2(inputs).transpose(1, 2) / self.
    sqrt_dim_k) @ self.v2(inputs)
        attention = self.attention_head_projection(torch.cat((head1, head2), 2))
        outputs = self.norm_mh(inputs + attention)
        #############################################################################
        #                               END OF YOUR CODE                            #
        #############################################################################
        return outputs

    def feedforward_layer(self, inputs):
        """
        :param inputs: float32 Tensor of shape (N,T,H)
        :returns outputs: float32 Tensor of shape (N,T,H)
        """
        # outputs = None
        #############################################################################
        # Deliverable 3: Implement the feedforward layer followed by add + norm.    #
        # Use a ReLU activation and apply the linear layers in the order you        #
        # initialized them.                                                         #
        # This should not take more than 3-5 lines of code.                         #
        #############################################################################
        outputs = self.norm_ff(inputs + self.ff_transform2(self.relu(self.ff_transform1(
    inputs))))
        #############################################################################
        #                               END OF YOUR CODE                            #
        #############################################################################
        return outputs

    def final_layer(self, inputs):
        """
        :param inputs: float32 Tensor of shape (N,T,H)
        :returns outputs: float32 Tensor of shape (N,1)
        """
        # outputs = None
        #############################################################################
        # Deliverable 4: Implement the final layer for the Transformer classifier.  #
        # This should not take more than 2 lines of code.                           #
        #############################################################################
```

```
        outputs = self.sigmoid(self.output_layer(inputs.reshape((inputs.shape[0], -1))))
        ##########################################################################
        #                                END OF YOUR CODE                        #
        ##########################################################################
        return outputs


def seed_torch(seed=0):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.backends.cudnn.benchmark = False
    torch.backends.cudnn.deterministic = True
```