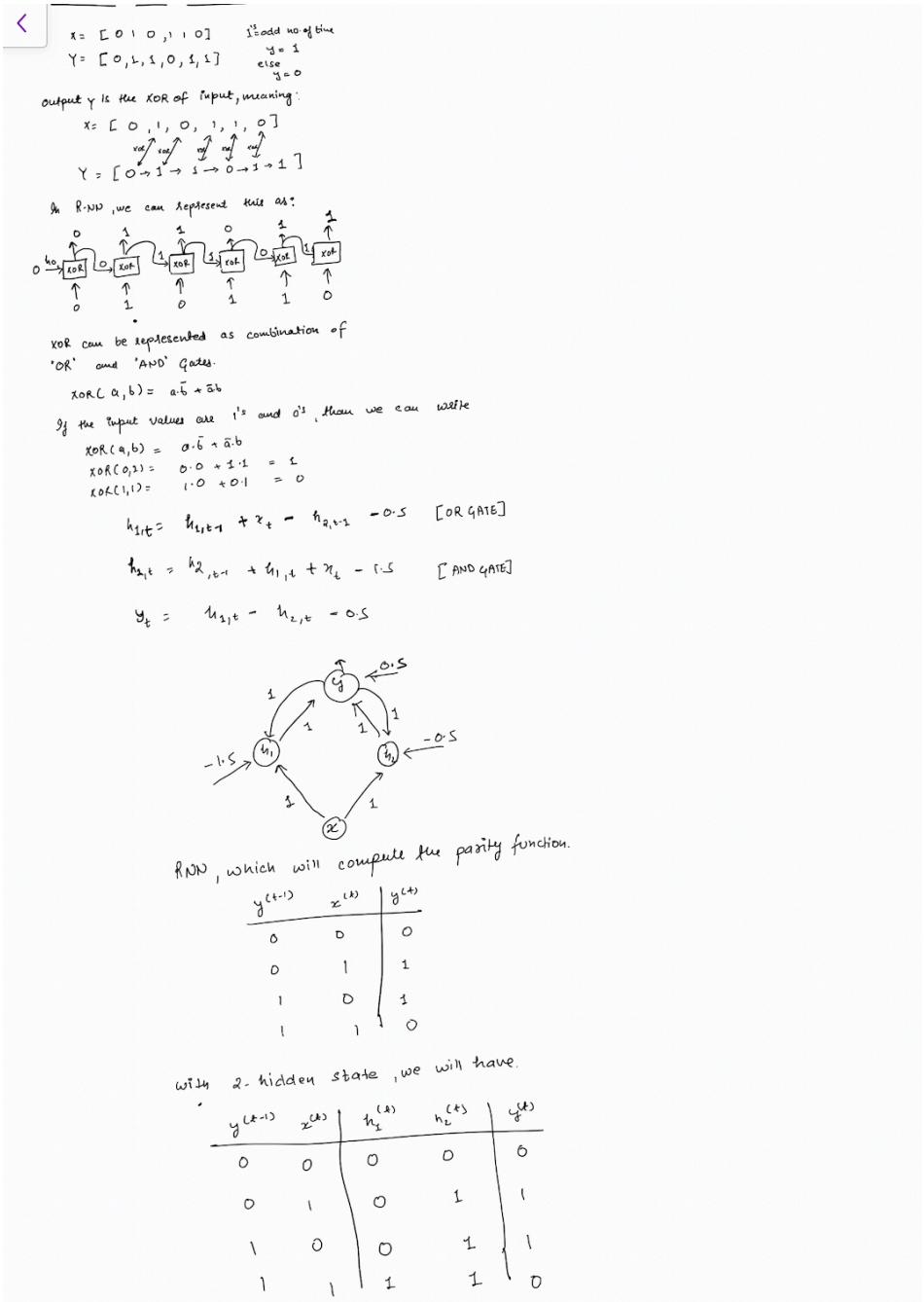
Deep Learning- HomeWork3

Gaurav Pande
gpande@gatech.edu

Question1: Recurrent Neural Network

1:Vanilla RNN for Parity Funtion-



2:LSTM for Parity Function-

② LSTM for Parity function :

Inputs :- h_{t-1}, x_t

output h_t

$h_t = \text{lstm\_XOR}(h_{t-1}, x_t)$

first we will set,

$C_t = h_t$, by setting the weights as

$$W_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$b_0 = 1$$

$$o_t = \sigma(W_0 [h_{t-1}, x_t] + b_0)$$

$$= \sigma(0 + 1)$$

$$= \sigma(1)$$

$$= 1$$

$$\left[\because \sigma(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases} \right]$$

$$h_t = o_t * \tanh(C_t)$$

$$= \tanh(C_t)$$

Now, if $C_t \in (0,1)$

$$h_t = C_t \quad \left[\because \tanh(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases} \right]$$

Now, we can also set

$\tilde{C}_t = 1 - h_t$, using

$$W_c = \begin{bmatrix} -1 \\ 0 \end{bmatrix}$$

$$b_c = 1$$

$$\tilde{C}_t = \tanh\left(\begin{bmatrix} -1 \\ 0 \end{bmatrix} (h_{t-1}, x_t) + 1 \right)$$

$$\tilde{C}_t = \tanh(1 - h_{t-1})$$

$$\tilde{C}_t = (1 - h_{t-1})$$

for $(1 - h_{t-1}) \in (0,1)$

Now we will want f_t to be 1 if $x_t = 0$, & otherwise

$$W_f = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

$$b_f = 1$$

$i_t = 1$ if x_t is 1 & otherwise

$$W_i = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$b_i = 0$$

Assumption : C_t and h_t are initialized to zero.

Another way to look at above drawn conclusion is: from the truth given

$$\text{xor}(x, y) = \bar{x}y + x\bar{y}$$

we can compare this with

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

$$C_t = f_t * (h_{t-1}) + i_t * (1 - h_{t-1})$$

f_t and i_t needs to be compliment of each other.

3: When to stop in Beam Search:

③: When to stop in a Beam Search:

Proof: If $B_{i,1} \leq best_{\leq i}$ then $B_{i,j} \leq B_{i,1} \leq best_{\leq i}$ for all items $B_{i,j}$ in the Beam B_i. future values will grow from these items will be no better, since probability ≤ 1, so all items in current and future steps are no better than $best_{\leq i}$.

4: Exploding Gradients:

④: Exploding Gradients :

$$h_t = w^T h_{t-1}$$
$$h_{t-1} = w^T h_{t-2}$$
$$h_{t-2} = w^T h_{t-3}$$
$$\vdots$$
$$h_1 = w^T h_0$$

putting values of $h_1, h_2, \dots h_{t-1}$ to get.

$$h_t = (w^T)^t h_0 \quad \text{—①}$$

As we know w is a square matrix, so we can do its Eigen Decomposition as:

$$w = P D P^{-1} \quad \text{—②}$$

where * D is the Diagnol matrix comprising of Diagnal values of w.

* P is the eigen vector of w

So, putting value of eq^n ② in eq^n ①.

$$h_t = \left((P D P^{-1})^T \right)^t h_0$$
$$h_t = \left((P^{-1})^T D^T P^T \right)^t h_0$$

[Using $(AB)^T = B^T A^T$]

$$h_t = (P^{-1})^T D P^T)^t h_0$$

[Using $\text{Diag}(A)^T = A$]

$$h_t = (((P^{-1})^T D P^T)(P^{-1})^T D P^T) \dots (P^{-1})^T D P^T)) h_0$$

we know $P^T \cdot (P^{-1})^T = I$, Hence.

$$h_t = ((P^{-1})^T D^t P^T)) h_0$$

Differentiating wrt h_0

$$\frac{dh_t}{dh_0} = (P^{-1})^T D^t P^T$$

Now, here, if $t \gg 0$, and if eigen value in $D < 1$, the elements of D^t will go to 0, resulting in vanishing gradient.

if eigen values in $D > 1$, then at least one value in D^t will go to ∞, resulting in the the exploding gradient.

RNN_Captioning

October 25, 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.

[56]:
```
# As usual, a bit of setup
from __future__ import print_function
import time, os, json
import numpy as np
import matplotlib.pyplot as plt

from cs231n.gradient_check import eval_numerical_gradient,␣
 ↪eval_numerical_gradient_array
from cs231n.rnn_layers import *
from cs231n.captioning_solver import CaptioningSolver
from cs231n.classifiers.rnn import CaptioningRNN
from cs231n.coco_utils import load_coco_data, sample_coco_minibatch,␣
 ↪decode_captions
from cs231n.image_utils import image_from_url

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

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

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

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

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:

[57]:
```
!pip install h5py
```

```
Requirement already satisfied: h5py in
/Users/gauravpande/Desktop/DL/AS_1/assignment/venv/lib/python3.7/site-packages
(2.10.0)
Requirement already satisfied: numpy>=1.7 in
/Users/gauravpande/Desktop/DL/AS_1/assignment/venv/lib/python3.7/site-packages
(from h5py) (1.17.1)
Requirement already satisfied: six in
/Users/gauravpande/Desktop/DL/AS_1/assignment/venv/lib/python3.7/site-packages
(from h5py) (1.12.0)
WARNING: You are using pip version 19.2.3, however version 19.3 is

available.

You should consider upgrading via the 'pip install --upgrade pip' command.
```

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:

[58]:
```
# 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**.

[59]:
```
# Sample a minibatch and show the images and captions
batch_size = 3

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

<START> a stop sign near a <UNK> <UNK> coffee shop <END>

<START> a bunch <UNK> food is pile <UNK> of the table <END>

<START> a bunch of vehicles sit in line as people walk by <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.

[60]:
```
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`.

[61]:
```
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:  2.319932372313319e-10
dprev_h error:  2.6828355645784327e-10
dWx error:  8.820244454238703e-10
dWh error:  4.703287554560559e-10
db error:  1.5956895526227225e-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`.

[62]:
```
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)
#print(x)
#print(x.shape)
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:  7.728466151011529e-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.

[63]:
```
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:  1.5402322184213243e-09
dh0 error:  3.3824326261334578e-09
dWx error:  7.238350796069372e-09
dWh error:  1.3157659173166636e-07
db error:  1.5353591509855146e-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 convert words (represented by integers) into vectors. Run the following to check your implementation. You should see error around `1e-8`.

[64]:
```
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`.

[65]:
```
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.

[66]:
```
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.

[67]:
```
# Sanity check for temporal softmax loss
from cs231n.rnn_layers import temporal_softmax_loss

N, T, V = 100, 1, 10

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

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

# Gradient check for temporal softmax loss
```

```
N, T, V = 7, 8, 9

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

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

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

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

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

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`.

[69]:
```
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.

[70]:
```
np.random.seed(231)

batch_size = 2
timesteps = 3
input_dim = 4
wordvec_dim = 5
hidden_dim = 6
word_to_idx = {'<NULL>': 0, 'cat': 2, 'dog': 3}
vocab_size = len(word_to_idx)

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

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

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

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

```
W_embed relative error: 2.331071e-09
W_proj relative error: 9.974426e-09
W_vocab relative error: 4.274378e-09
```

```
Wh relative error: 5.557955e-09
Wx relative error: 7.725620e-07
b relative error: 8.001353e-10
b_proj relative error: 6.260039e-09
b_vocab relative error: 1.690334e-09
```

13 Overfit small data

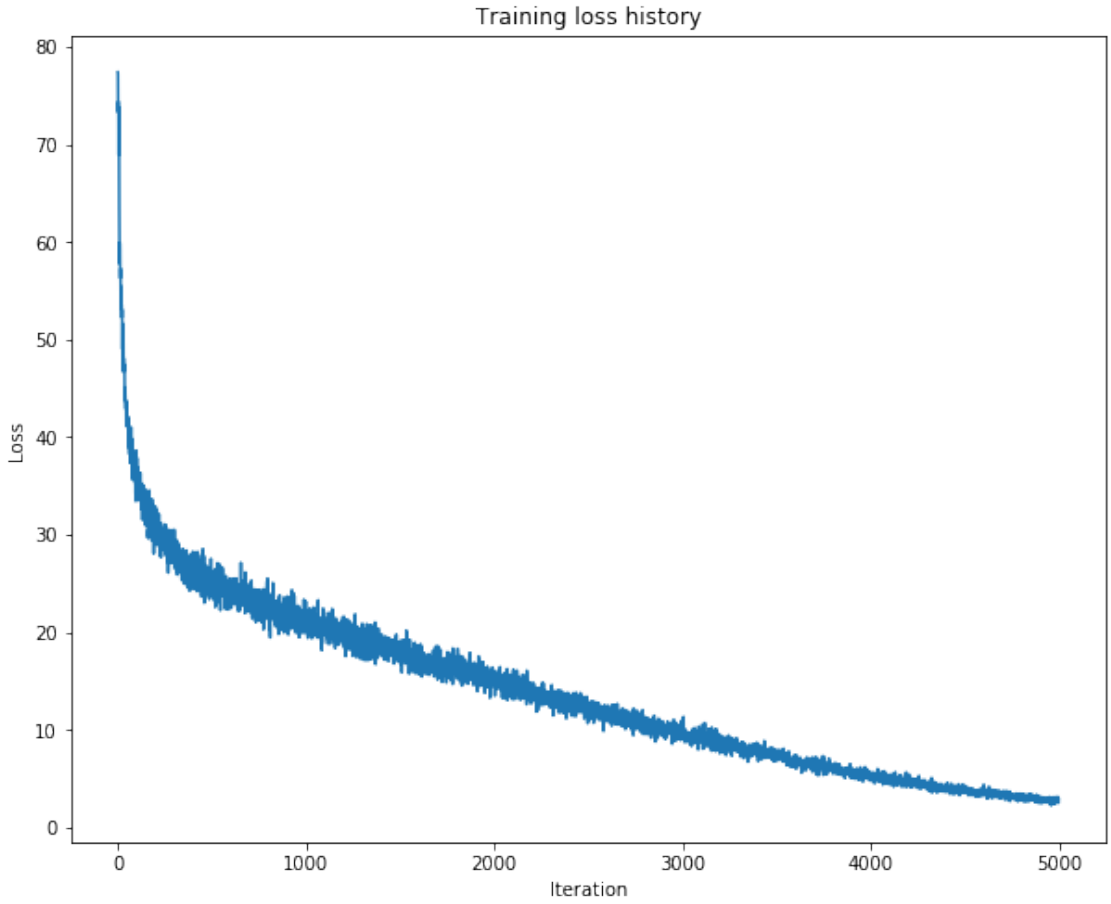
Similar to the `Solver` class that we used to train image classification models on the previous assignment, on this assignment we use a `CaptioningSolver` class to train image captioning models. Open the file `cs231n/captioning_solver.py` and read through the `CaptioningSolver` class; it should look 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.

[71]:
```
np.random.seed(231)

small_data = load_coco_data(max_train=10000)

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

small_rnn_solver = CaptioningSolver(small_rnn_model, small_data,
           update_rule='adam',
           num_epochs=50,
           batch_size=100,
           optim_config={
             'learning_rate': 5e-3,
           },
           lr_decay=0.95,
           verbose=True, print_every=10,
         )

small_rnn_solver.train()

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

```
(Iteration 1 / 5000) loss: 77.409794
(Iteration 11 / 5000) loss: 57.753746
(Iteration 21 / 5000) loss: 53.169962
(Iteration 31 / 5000) loss: 50.221674
(Iteration 41 / 5000) loss: 44.573924
(Iteration 51 / 5000) loss: 41.694370
(Iteration 61 / 5000) loss: 40.804080
(Iteration 71 / 5000) loss: 39.487258
(Iteration 81 / 5000) loss: 36.869255
(Iteration 91 / 5000) loss: 37.817425
(Iteration 101 / 5000) loss: 36.721799
(Iteration 111 / 5000) loss: 34.242719
(Iteration 121 / 5000) loss: 36.355007
(Iteration 131 / 5000) loss: 33.818252
(Iteration 141 / 5000) loss: 33.678754
(Iteration 151 / 5000) loss: 30.922859
(Iteration 161 / 5000) loss: 29.729601
(Iteration 171 / 5000) loss: 31.477432
(Iteration 181 / 5000) loss: 33.173368
(Iteration 191 / 5000) loss: 31.400335
(Iteration 201 / 5000) loss: 32.895437
(Iteration 211 / 5000) loss: 31.211643
(Iteration 221 / 5000) loss: 29.618174
(Iteration 231 / 5000) loss: 27.705580
(Iteration 241 / 5000) loss: 29.738037
(Iteration 251 / 5000) loss: 31.068388
(Iteration 261 / 5000) loss: 28.513412
(Iteration 271 / 5000) loss: 28.545572
(Iteration 281 / 5000) loss: 29.482339
(Iteration 291 / 5000) loss: 29.645118
(Iteration 301 / 5000) loss: 30.520204
(Iteration 311 / 5000) loss: 27.869443
(Iteration 321 / 5000) loss: 27.234216
(Iteration 331 / 5000) loss: 27.903267
(Iteration 341 / 5000) loss: 25.356266
(Iteration 351 / 5000) loss: 25.111743
(Iteration 361 / 5000) loss: 26.052928
(Iteration 371 / 5000) loss: 26.019760
(Iteration 381 / 5000) loss: 27.712689
(Iteration 391 / 5000) loss: 28.057124
(Iteration 401 / 5000) loss: 28.280853
(Iteration 411 / 5000) loss: 28.157169
(Iteration 421 / 5000) loss: 25.542835
(Iteration 431 / 5000) loss: 26.217914
(Iteration 441 / 5000) loss: 26.679728
(Iteration 451 / 5000) loss: 27.979191
(Iteration 461 / 5000) loss: 23.028289
(Iteration 471 / 5000) loss: 26.196959
```

```
(Iteration 4801 / 5000) loss: 3.335666
(Iteration 4811 / 5000) loss: 2.962112
(Iteration 4821 / 5000) loss: 3.109034
(Iteration 4831 / 5000) loss: 2.829207
(Iteration 4841 / 5000) loss: 2.990116
(Iteration 4851 / 5000) loss: 3.068530
(Iteration 4861 / 5000) loss: 3.129015
(Iteration 4871 / 5000) loss: 3.139111
(Iteration 4881 / 5000) loss: 3.033974
(Iteration 4891 / 5000) loss: 2.542617
(Iteration 4901 / 5000) loss: 2.821543
(Iteration 4911 / 5000) loss: 2.661179
(Iteration 4921 / 5000) loss: 2.976442
(Iteration 4931 / 5000) loss: 2.702280
(Iteration 4941 / 5000) loss: 2.666148
(Iteration 4951 / 5000) loss: 3.085675
(Iteration 4961 / 5000) loss: 2.710505
(Iteration 4971 / 5000) loss: 2.973529
(Iteration 4981 / 5000) loss: 2.599850
(Iteration 4991 / 5000) loss: 2.998803
```



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.

[80]:
```
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 picture of an old boat with a lot of people around it <END>
GT:<START> a picture of an old boat with a lot of people around it <END>

train
a group of sheep is standing in a field of grass <END>
GT:<START> a group of sheep is standing in a field of grass <END>

val
a man with a motorcycle parked in front of a building <END>
GT:<START> a motorcycle parked in a pile of rocks <END>

val
an orange and an orange <UNK> is next to a bowl of food <END>
GT:<START> two bottles of <UNK> on a table with bowl of <UNK> <END>

[ ]:

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_numerica
l_gradient_array
from cs231n.rnn_layers import *
from cs231n.captioning_solver import CaptioningSolver
from cs231n.classifiers.rnn import CaptioningRNN
from cs231n.coco_utils import load_coco_data, sample_coco_minibatch, dec
ode_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-i
n-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
))))
```

In [10]:
```
!pip install nltk
```

```
DEPRECATION: Python 2.7 will reach the end of its life on January 1st,
 2020. Please upgrade your Python as Python 2.7 won't be maintained aft
er that date. A future version of pip will drop support for Python 2.7.
Requirement already satisfied: nltk in /usr/local/lib/python2.7/site-pa
ckages (3.4.5)
Requirement already satisfied: six in /usr/local/lib/python2.7/site-pac
kages (from nltk) (1.12.0)
Requirement already satisfied: singledispatch; python_version < "3.4" i
n /usr/local/lib/python2.7/site-packages (from nltk) (3.4.0.3)
```

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 bel
ow.
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
```

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

$$i = \sigma(a_i) \qquad f = \sigma(a_f) \qquad o = \sigma(a_o) \qquad g = \tanh(a_g)$$

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

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

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 $A = X_t W_x + H_{t-1} W_h$

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:  5.7054131185818695e-09
next_c error:  5.8143123088804145e-09
```

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:  6.141372523522652e-10
dh error:  3.0914746081903265e-10
dc error:  1.5221795880129153e-10
dWx error:  1.6933646448886643e-09
dWh error:  4.806248540056623e-08
db error:  1.734924139321044e-10
```

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
```

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:  6.186232096167152e-09
dh0 error:  6.791883730936431e-09
dWx error:  3.301449507829467e-09
dWh error:  1.5077886468004443e-06
db error:  7.214753825317872e-10
```

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))
```

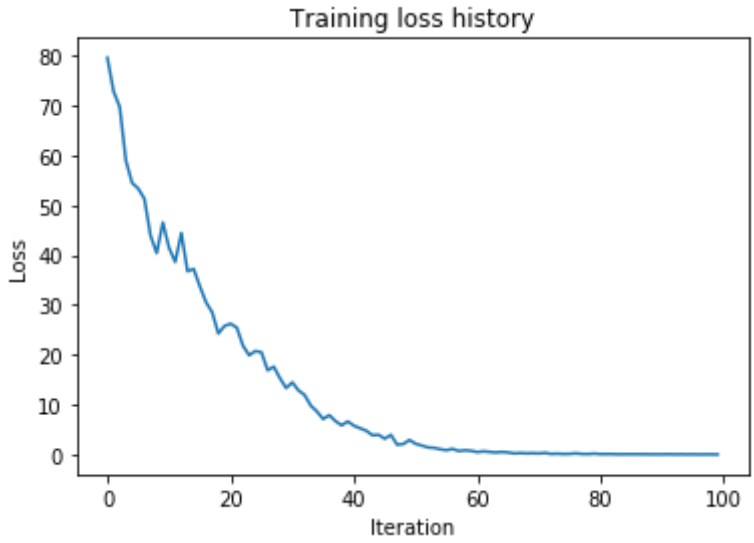
```
loss:  9.824459354432264
expected loss:  9.82445935443
difference:  2.2648549702353193e-12
```

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 [9]:
```
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
Average BLEU score for train: 0.202956
Average BLEU score for val: 0.076891
(Iteration 11 / 100) loss: 41.479142
Average BLEU score for train: 0.202854
Average BLEU score for val: 0.110803
(Iteration 21 / 100) loss: 26.261340
Average BLEU score for train: 0.398885
Average BLEU score for val: 0.094894
(Iteration 31 / 100) loss: 14.521850
Average BLEU score for train: 0.714101
Average BLEU score for val: 0.112676
(Iteration 41 / 100) loss: 5.829343
Average BLEU score for train: 0.919866
Average BLEU score for val: 0.130647
(Iteration 51 / 100) loss: 2.221016
Average BLEU score for train: 0.976698
Average BLEU score for val: 0.141936
(Iteration 61 / 100) loss: 0.556898
Average BLEU score for train: 0.994739
Average BLEU score for val: 0.154023
(Iteration 71 / 100) loss: 0.285582
Average BLEU score for train: 1.000000
Average BLEU score for val: 0.138560
(Iteration 81 / 100) loss: 0.150890
Average BLEU score for train: 1.000000
Average BLEU score for val: 0.149335
(Iteration 91 / 100) loss: 0.108281
Average BLEU score for train: 1.000000
Average BLEU score for val: 0.144635
```



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.

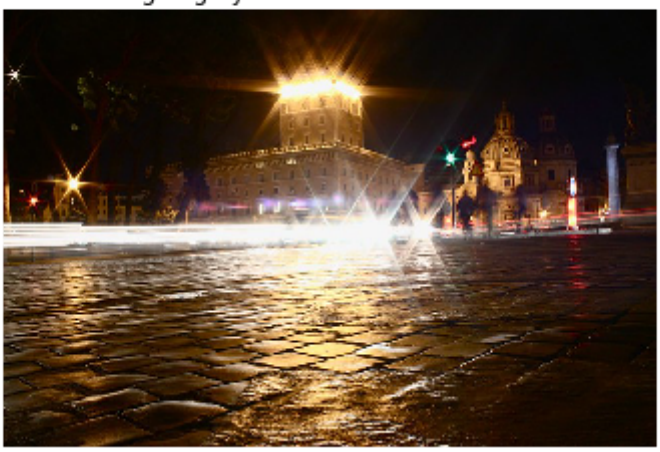
In [10]:
```
for split in ['train', 'val']:
    minibatch = sample_coco_minibatch(small_data, split=split, batch_siz
e=2)
    gt_captions, features, urls = minibatch
    gt_captions = decode_captions(gt_captions, data['idx_to_word'])

    sample_captions = small_lstm_model.sample(features)
    sample_captions = decode_captions(sample_captions, data['idx_to_wor
d'])

    for gt_caption, sample_caption, url in zip(gt_captions, sample_capti
ons, 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
two of the <UNK> <UNK> during a game <END>
GT:<START> two of the <UNK> <UNK> during a game <END>

train
half a <UNK> <UNK> out on the ocean <END>
GT:<START> half a <UNK> <UNK> out on the ocean <END>

val
a small with a donut in his hand <END>
GT:<START> a young boy eating a big slice of pizza <END>

val
a group of a <UNK> <END>
GT:<START> traffic going by <UNK> <UNK> it is like a <UNK> <END>



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 (http://www.aclweb.org/anthology/P02-1040.pdf) 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 [17]:
```
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, weights = [1])
    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]))
```

write a description of your model here:

Please view the second last iteration(Iteration 1471 / 16650) for correct BLEU score

This part of code took a lot of time to figure out how to do it. I implemented it correctly before but my sample method in CaptionRNN class was wrongly implemented, so i could see my loss going doing but then BLEU score was not changing a bit. So i tested the BLEU score first in the overfit model and thats is where i figure out my method in captionRNN is wrong. After fixing it, i could see that in my overfit part, the train BLEU score was 1 and valid score was 0.01, which makes perfect sense(see the overfit section above)

I implemented the evaluate function for Bleu score in the captionsolver class under train method to see the bleu score after every 10th iteration to see how well my hyperparameters are doing. This way i was able to quickly analyze when my model is overfitting and when it is underfitting. Initially the model overfits and my BleU score doesnt do well, but then I started changing learning parameter, but that also didnt help. I realize I need to increase my traning size, So then i increase the max_train size to 100000, because the model needs to look at all the images.

Final with 100000 of max_train size, and learning rate of 2e-3 gave me the results after long iterations. I exit the iteratinons the moment i get BLEU score for validation and test > 0.3

As you can see the second last is the one. Is should have store all the values in list and returned parameters, but the training in my mac takes so much time, that is why i had to interrupt in between.

Average BLEU score for train: 0.328226

Average BLEU score for val: 0.307793

In [11]:
```
# write your code to train your model here.
# make sure to include the call to evaluate_model which prints out your
 highest validation BLEU score.
# data = load_coco_data(pca_features=True)
np.random.seed(231)
small_data2 = load_coco_data(max_train=100000)
good_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 )

good_lstm_solver = CaptioningSolver(good_lstm_model, small_data2,
           update_rule='adam',
           num_epochs=50,
           batch_size=300,
           optim_config={
             'learning_rate': 2e-3,
           },
           lr_decay=0.995,
           verbose=True, print_every=10,
         )

good_lstm_solver.train()
```

```
(Iteration 1 / 16650) loss: 76.512785
Average BLEU score for train: 0.046964
Average BLEU score for val: 0.044041
(Iteration 11 / 16650) loss: 68.077276
Average BLEU score for train: 0.054750
Average BLEU score for val: 0.053018
(Iteration 21 / 16650) loss: 54.645203
Average BLEU score for train: 0.117009
Average BLEU score for val: 0.114206
(Iteration 31 / 16650) loss: 51.163827
Average BLEU score for train: 0.114474
Average BLEU score for val: 0.111853
(Iteration 41 / 16650) loss: 50.287091
Average BLEU score for train: 0.104094
Average BLEU score for val: 0.103355
(Iteration 51 / 16650) loss: 49.102843
Average BLEU score for train: 0.014282
Average BLEU score for val: 0.015127
(Iteration 61 / 16650) loss: 46.130155
Average BLEU score for train: 0.043858
Average BLEU score for val: 0.049103
(Iteration 71 / 16650) loss: 45.115773
Average BLEU score for train: 0.094845
Average BLEU score for val: 0.090462
(Iteration 81 / 16650) loss: 44.005203
Average BLEU score for train: 0.121996
Average BLEU score for val: 0.120573
(Iteration 91 / 16650) loss: 42.274486
Average BLEU score for train: 0.187782
Average BLEU score for val: 0.185683
(Iteration 101 / 16650) loss: 41.510075
Average BLEU score for train: 0.177020
Average BLEU score for val: 0.174047
(Iteration 111 / 16650) loss: 39.362343
Average BLEU score for train: 0.174061
Average BLEU score for val: 0.164294
(Iteration 121 / 16650) loss: 37.871784
Average BLEU score for train: 0.154966
Average BLEU score for val: 0.150972
(Iteration 131 / 16650) loss: 36.765185
Average BLEU score for train: 0.152451
Average BLEU score for val: 0.152683
(Iteration 141 / 16650) loss: 36.578958
Average BLEU score for train: 0.132366
Average BLEU score for val: 0.126333
(Iteration 151 / 16650) loss: 36.221491
Average BLEU score for train: 0.134765
Average BLEU score for val: 0.145524
(Iteration 161 / 16650) loss: 36.172265
Average BLEU score for train: 0.173005
Average BLEU score for val: 0.171139
(Iteration 171 / 16650) loss: 34.059236
Average BLEU score for train: 0.193297
Average BLEU score for val: 0.185583
(Iteration 181 / 16650) loss: 34.997861
Average BLEU score for train: 0.204931
Average BLEU score for val: 0.191778
```

```
(Iteration 191 / 16650) loss: 34.708261
Average BLEU score for train: 0.200519
Average BLEU score for val: 0.205494
(Iteration 201 / 16650) loss: 34.009456
Average BLEU score for train: 0.214451
Average BLEU score for val: 0.212404
(Iteration 211 / 16650) loss: 33.692329
Average BLEU score for train: 0.207267
Average BLEU score for val: 0.211423
(Iteration 221 / 16650) loss: 32.774109
Average BLEU score for train: 0.209365
Average BLEU score for val: 0.208031
(Iteration 231 / 16650) loss: 32.725878
Average BLEU score for train: 0.226052
Average BLEU score for val: 0.227909
(Iteration 241 / 16650) loss: 31.194724
Average BLEU score for train: 0.212305
Average BLEU score for val: 0.220214
(Iteration 251 / 16650) loss: 32.465865
Average BLEU score for train: 0.246445
Average BLEU score for val: 0.243367
(Iteration 261 / 16650) loss: 31.788001
Average BLEU score for train: 0.203439
Average BLEU score for val: 0.213576
(Iteration 271 / 16650) loss: 31.882635
Average BLEU score for train: 0.234973
Average BLEU score for val: 0.223165
(Iteration 281 / 16650) loss: 31.257389
Average BLEU score for train: 0.261446
Average BLEU score for val: 0.250903
(Iteration 291 / 16650) loss: 30.477113
Average BLEU score for train: 0.268198
Average BLEU score for val: 0.261863
(Iteration 301 / 16650) loss: 31.157326
Average BLEU score for train: 0.244106
Average BLEU score for val: 0.228961
(Iteration 311 / 16650) loss: 30.809403
Average BLEU score for train: 0.245743
Average BLEU score for val: 0.227699
(Iteration 321 / 16650) loss: 29.741073
Average BLEU score for train: 0.224877
Average BLEU score for val: 0.221272
(Iteration 331 / 16650) loss: 30.615574
Average BLEU score for train: 0.253172
Average BLEU score for val: 0.259019
(Iteration 341 / 16650) loss: 29.470562
Average BLEU score for train: 0.251057
Average BLEU score for val: 0.249879
(Iteration 351 / 16650) loss: 29.752151
Average BLEU score for train: 0.260834
Average BLEU score for val: 0.250512
(Iteration 361 / 16650) loss: 29.933636
Average BLEU score for train: 0.263261
Average BLEU score for val: 0.244279
(Iteration 371 / 16650) loss: 30.045493
Average BLEU score for train: 0.248926
Average BLEU score for val: 0.241407
```

```
(Iteration 381 / 16650) loss: 29.121295
Average BLEU score for train: 0.285594
Average BLEU score for val: 0.266838
(Iteration 391 / 16650) loss: 29.643437
Average BLEU score for train: 0.264281
Average BLEU score for val: 0.252219
(Iteration 401 / 16650) loss: 28.937667
Average BLEU score for train: 0.265720
Average BLEU score for val: 0.255684
(Iteration 411 / 16650) loss: 29.312577
Average BLEU score for train: 0.253085
Average BLEU score for val: 0.249310
(Iteration 421 / 16650) loss: 28.827288
Average BLEU score for train: 0.258559
Average BLEU score for val: 0.239826
(Iteration 431 / 16650) loss: 28.730817
Average BLEU score for train: 0.262461
Average BLEU score for val: 0.255888
(Iteration 441 / 16650) loss: 28.138362
Average BLEU score for train: 0.257080
Average BLEU score for val: 0.247702
(Iteration 451 / 16650) loss: 29.174410
Average BLEU score for train: 0.270191
Average BLEU score for val: 0.255502
(Iteration 461 / 16650) loss: 29.415328
Average BLEU score for train: 0.250076
Average BLEU score for val: 0.239469
(Iteration 471 / 16650) loss: 28.722077
Average BLEU score for train: 0.264817
Average BLEU score for val: 0.244250
(Iteration 481 / 16650) loss: 27.198391
Average BLEU score for train: 0.267481
Average BLEU score for val: 0.257967
(Iteration 491 / 16650) loss: 28.147840
Average BLEU score for train: 0.276901
Average BLEU score for val: 0.264037
(Iteration 501 / 16650) loss: 29.195641
Average BLEU score for train: 0.252731
Average BLEU score for val: 0.244778
(Iteration 511 / 16650) loss: 27.732254
Average BLEU score for train: 0.261791
Average BLEU score for val: 0.261882
(Iteration 521 / 16650) loss: 27.313178
Average BLEU score for train: 0.269059
Average BLEU score for val: 0.267816
(Iteration 531 / 16650) loss: 28.091101
Average BLEU score for train: 0.287489
Average BLEU score for val: 0.274228
(Iteration 541 / 16650) loss: 28.993446
Average BLEU score for train: 0.273235
Average BLEU score for val: 0.262128
(Iteration 551 / 16650) loss: 27.427098
Average BLEU score for train: 0.272962
Average BLEU score for val: 0.264414
(Iteration 561 / 16650) loss: 27.835226
Average BLEU score for train: 0.288447
Average BLEU score for val: 0.272242
```

```
(Iteration 571 / 16650) loss: 28.029334
Average BLEU score for train: 0.247163
Average BLEU score for val: 0.239308
(Iteration 581 / 16650) loss: 27.443562
Average BLEU score for train: 0.256061
Average BLEU score for val: 0.243522
(Iteration 591 / 16650) loss: 28.187435
Average BLEU score for train: 0.290790
Average BLEU score for val: 0.278516
(Iteration 601 / 16650) loss: 27.794266
Average BLEU score for train: 0.283654
Average BLEU score for val: 0.267898
(Iteration 611 / 16650) loss: 27.625552
Average BLEU score for train: 0.275560
Average BLEU score for val: 0.268871
(Iteration 621 / 16650) loss: 27.366096
Average BLEU score for train: 0.289256
Average BLEU score for val: 0.291139
(Iteration 631 / 16650) loss: 27.762143
Average BLEU score for train: 0.283462
Average BLEU score for val: 0.269859
(Iteration 641 / 16650) loss: 26.851792
Average BLEU score for train: 0.289038
Average BLEU score for val: 0.273387
(Iteration 651 / 16650) loss: 26.620749
Average BLEU score for train: 0.267196
Average BLEU score for val: 0.255646
(Iteration 661 / 16650) loss: 26.141276
Average BLEU score for train: 0.293187
Average BLEU score for val: 0.282751
(Iteration 671 / 16650) loss: 26.606734
Average BLEU score for train: 0.265458
Average BLEU score for val: 0.264036
(Iteration 681 / 16650) loss: 26.782038
Average BLEU score for train: 0.288463
Average BLEU score for val: 0.280768
(Iteration 691 / 16650) loss: 26.477262
Average BLEU score for train: 0.270235
Average BLEU score for val: 0.258241
(Iteration 701 / 16650) loss: 26.682881
Average BLEU score for train: 0.283981
Average BLEU score for val: 0.276632
(Iteration 711 / 16650) loss: 26.731201
Average BLEU score for train: 0.286662
Average BLEU score for val: 0.272375
(Iteration 721 / 16650) loss: 27.290718
Average BLEU score for train: 0.285684
Average BLEU score for val: 0.275586
(Iteration 731 / 16650) loss: 26.415401
Average BLEU score for train: 0.285096
Average BLEU score for val: 0.270485
(Iteration 741 / 16650) loss: 26.509080
Average BLEU score for train: 0.299761
Average BLEU score for val: 0.295322
(Iteration 751 / 16650) loss: 26.487593
Average BLEU score for train: 0.305595
Average BLEU score for val: 0.281193
```

```
(Iteration 761 / 16650) loss: 25.885875
Average BLEU score for train: 0.286386
Average BLEU score for val: 0.272163
(Iteration 771 / 16650) loss: 25.518479
Average BLEU score for train: 0.315466
Average BLEU score for val: 0.284466
(Iteration 781 / 16650) loss: 26.511946
Average BLEU score for train: 0.289504
Average BLEU score for val: 0.272574
(Iteration 791 / 16650) loss: 26.092233
Average BLEU score for train: 0.289345
Average BLEU score for val: 0.273749
(Iteration 801 / 16650) loss: 26.415936
Average BLEU score for train: 0.288863
Average BLEU score for val: 0.274498
(Iteration 811 / 16650) loss: 26.828772
Average BLEU score for train: 0.283588
Average BLEU score for val: 0.260834
(Iteration 821 / 16650) loss: 26.863449
Average BLEU score for train: 0.280796
Average BLEU score for val: 0.255958
(Iteration 831 / 16650) loss: 25.563962
Average BLEU score for train: 0.278402
Average BLEU score for val: 0.259077
(Iteration 841 / 16650) loss: 26.851231
Average BLEU score for train: 0.291008
Average BLEU score for val: 0.284601
(Iteration 851 / 16650) loss: 26.312038
Average BLEU score for train: 0.288932
Average BLEU score for val: 0.281121
(Iteration 861 / 16650) loss: 26.604260
Average BLEU score for train: 0.281712
Average BLEU score for val: 0.264834
(Iteration 871 / 16650) loss: 26.132024
Average BLEU score for train: 0.297812
Average BLEU score for val: 0.272212
(Iteration 881 / 16650) loss: 25.905669
Average BLEU score for train: 0.317681
Average BLEU score for val: 0.278328
(Iteration 891 / 16650) loss: 26.912422
Average BLEU score for train: 0.292581
Average BLEU score for val: 0.266838
(Iteration 901 / 16650) loss: 25.886046
Average BLEU score for train: 0.319359
Average BLEU score for val: 0.287139
(Iteration 911 / 16650) loss: 25.285070
Average BLEU score for train: 0.302570
Average BLEU score for val: 0.276099
(Iteration 921 / 16650) loss: 24.962257
Average BLEU score for train: 0.296592
Average BLEU score for val: 0.270665
(Iteration 931 / 16650) loss: 25.072538
Average BLEU score for train: 0.284320
Average BLEU score for val: 0.275283
(Iteration 941 / 16650) loss: 24.834605
Average BLEU score for train: 0.298439
Average BLEU score for val: 0.280659
```

```
(Iteration 951 / 16650) loss: 24.621629
Average BLEU score for train: 0.279621
Average BLEU score for val: 0.257019
(Iteration 961 / 16650) loss: 26.412811
Average BLEU score for train: 0.301058
Average BLEU score for val: 0.278556
(Iteration 971 / 16650) loss: 24.979272
Average BLEU score for train: 0.297493
Average BLEU score for val: 0.282414
(Iteration 981 / 16650) loss: 25.333525
Average BLEU score for train: 0.308021
Average BLEU score for val: 0.287757
(Iteration 991 / 16650) loss: 25.749301
Average BLEU score for train: 0.308477
Average BLEU score for val: 0.295902
(Iteration 1001 / 16650) loss: 25.204308
Average BLEU score for train: 0.295851
Average BLEU score for val: 0.270025
(Iteration 1011 / 16650) loss: 24.650243
Average BLEU score for train: 0.296078
Average BLEU score for val: 0.278397
(Iteration 1021 / 16650) loss: 25.148533
Average BLEU score for train: 0.321915
Average BLEU score for val: 0.287691
(Iteration 1031 / 16650) loss: 25.965568
Average BLEU score for train: 0.292790
Average BLEU score for val: 0.275594
(Iteration 1041 / 16650) loss: 25.869901
Average BLEU score for train: 0.307681
Average BLEU score for val: 0.286560
(Iteration 1051 / 16650) loss: 25.706667
Average BLEU score for train: 0.305454
Average BLEU score for val: 0.282265
(Iteration 1061 / 16650) loss: 24.407620
Average BLEU score for train: 0.309035
Average BLEU score for val: 0.275262
(Iteration 1071 / 16650) loss: 23.854423
Average BLEU score for train: 0.297478
Average BLEU score for val: 0.266439
(Iteration 1081 / 16650) loss: 25.227175
Average BLEU score for train: 0.286934
Average BLEU score for val: 0.275822
(Iteration 1091 / 16650) loss: 24.700606
Average BLEU score for train: 0.293405
Average BLEU score for val: 0.275755
(Iteration 1101 / 16650) loss: 24.849424
Average BLEU score for train: 0.314086
Average BLEU score for val: 0.284349
(Iteration 1111 / 16650) loss: 24.241071
Average BLEU score for train: 0.303646
Average BLEU score for val: 0.273424
(Iteration 1121 / 16650) loss: 25.272117
Average BLEU score for train: 0.301661
Average BLEU score for val: 0.280944
(Iteration 1131 / 16650) loss: 24.481606
Average BLEU score for train: 0.298306
Average BLEU score for val: 0.275362
```

```
(Iteration 1141 / 16650) loss: 25.257430
Average BLEU score for train: 0.326245
Average BLEU score for val: 0.301026
(Iteration 1151 / 16650) loss: 24.660716
Average BLEU score for train: 0.305642
Average BLEU score for val: 0.270922
(Iteration 1161 / 16650) loss: 25.075980
Average BLEU score for train: 0.313694
Average BLEU score for val: 0.299387
(Iteration 1171 / 16650) loss: 23.524729
Average BLEU score for train: 0.303839
Average BLEU score for val: 0.284561
(Iteration 1181 / 16650) loss: 24.701108
Average BLEU score for train: 0.315548
Average BLEU score for val: 0.289585
(Iteration 1191 / 16650) loss: 24.768657
Average BLEU score for train: 0.319544
Average BLEU score for val: 0.290135
(Iteration 1201 / 16650) loss: 24.581477
Average BLEU score for train: 0.297748
Average BLEU score for val: 0.260141
(Iteration 1211 / 16650) loss: 24.785977
Average BLEU score for train: 0.308835
Average BLEU score for val: 0.280186
(Iteration 1221 / 16650) loss: 24.939197
Average BLEU score for train: 0.304318
Average BLEU score for val: 0.290947
(Iteration 1231 / 16650) loss: 24.078460
Average BLEU score for train: 0.315076
Average BLEU score for val: 0.281375
(Iteration 1241 / 16650) loss: 24.665725
Average BLEU score for train: 0.306246
Average BLEU score for val: 0.273203
(Iteration 1251 / 16650) loss: 24.612927
Average BLEU score for train: 0.320902
Average BLEU score for val: 0.286828
(Iteration 1261 / 16650) loss: 24.732859
Average BLEU score for train: 0.299962
Average BLEU score for val: 0.277798
(Iteration 1271 / 16650) loss: 23.645640
Average BLEU score for train: 0.309723
Average BLEU score for val: 0.289697
(Iteration 1281 / 16650) loss: 24.282888
Average BLEU score for train: 0.308511
Average BLEU score for val: 0.281045
(Iteration 1291 / 16650) loss: 24.531189
Average BLEU score for train: 0.317929
Average BLEU score for val: 0.290503
(Iteration 1301 / 16650) loss: 24.416990
Average BLEU score for train: 0.307649
Average BLEU score for val: 0.270123
(Iteration 1311 / 16650) loss: 24.420007
Average BLEU score for train: 0.321966
Average BLEU score for val: 0.289109
(Iteration 1321 / 16650) loss: 24.763796
Average BLEU score for train: 0.328848
Average BLEU score for val: 0.288638
```

```
(Iteration 1331 / 16650) loss: 24.963264
Average BLEU score for train: 0.299862
Average BLEU score for val: 0.271481
(Iteration 1341 / 16650) loss: 24.305354
Average BLEU score for train: 0.306032
Average BLEU score for val: 0.298111
(Iteration 1351 / 16650) loss: 23.478862
Average BLEU score for train: 0.320525
Average BLEU score for val: 0.293046
(Iteration 1361 / 16650) loss: 24.647107
Average BLEU score for train: 0.305598
Average BLEU score for val: 0.286300
(Iteration 1371 / 16650) loss: 24.174549
Average BLEU score for train: 0.318598
Average BLEU score for val: 0.272790
(Iteration 1381 / 16650) loss: 24.316431
Average BLEU score for train: 0.308642
Average BLEU score for val: 0.290677
(Iteration 1391 / 16650) loss: 24.782471
Average BLEU score for train: 0.318778
Average BLEU score for val: 0.285458
(Iteration 1401 / 16650) loss: 23.021588
Average BLEU score for train: 0.314646
Average BLEU score for val: 0.284953
(Iteration 1411 / 16650) loss: 23.696909
Average BLEU score for train: 0.316307
Average BLEU score for val: 0.294044
(Iteration 1421 / 16650) loss: 24.057474
Average BLEU score for train: 0.321968
Average BLEU score for val: 0.289745
(Iteration 1431 / 16650) loss: 23.760354
Average BLEU score for train: 0.309414
Average BLEU score for val: 0.282944
(Iteration 1441 / 16650) loss: 22.920734
Average BLEU score for train: 0.322910
Average BLEU score for val: 0.292970
(Iteration 1451 / 16650) loss: 23.795938
Average BLEU score for train: 0.321821
Average BLEU score for val: 0.288846
(Iteration 1461 / 16650) loss: 23.669206
Average BLEU score for train: 0.332836
Average BLEU score for val: 0.299062
(Iteration 1471 / 16650) loss: 24.090505
Average BLEU score for train: 0.328226
Average BLEU score for val: 0.307793
(Iteration 1481 / 16650) loss: 23.746292
Average BLEU score for train: 0.303657
Average BLEU score for val: 0.278762
```

Transformer_Classification

October 25, 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").

[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.

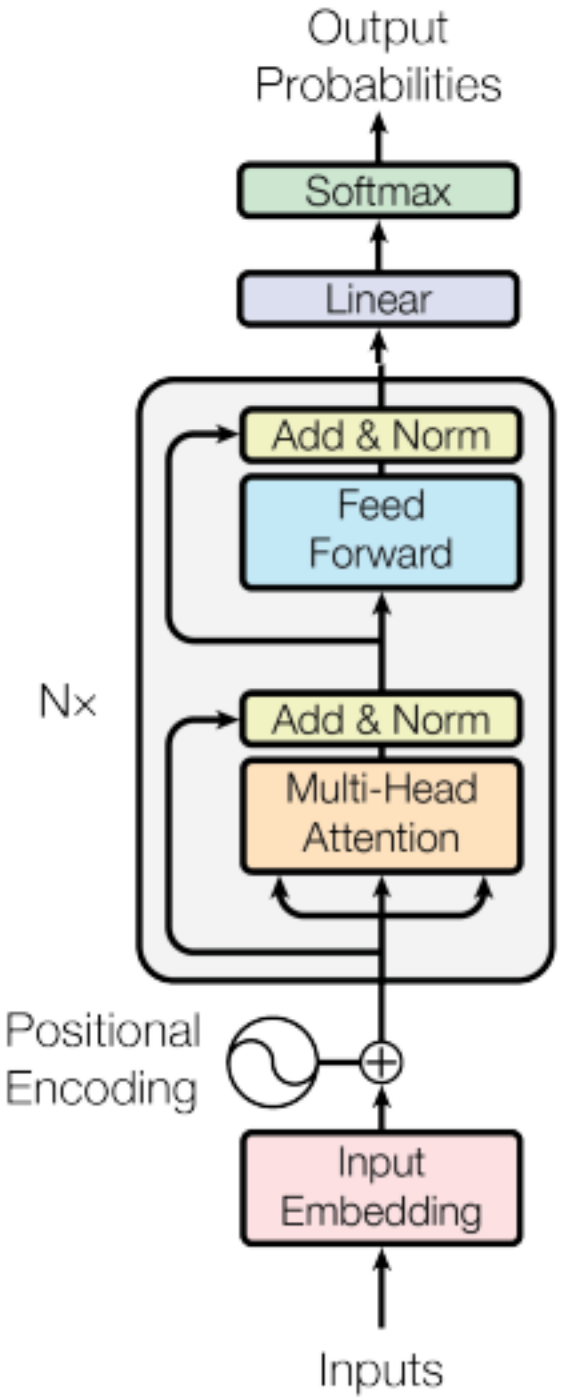
We've additionally already constructed a vocabulary and converted all of the strings of tokens into integers which can be used for vocabulary lookup for you. Feel free to explore the data here.

[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:



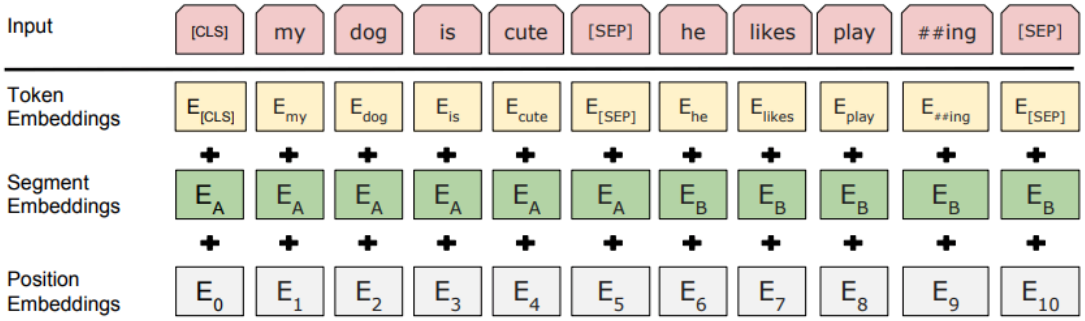
You can refer to the original paper for more details.

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.

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`.

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

model = ClassificationTransformer(word_to_ix, hidden_dim=128, num_heads=2,␣
↪dim_feedforward=2048, dim_k=96,
                                  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())␣
↪# should be very small (<0.01)
except:
    print("NOT IMPLEMENTED")
```

```
Difference: 0.0017988268518820405
```

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.

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.

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

try:
    print("Difference:", torch.sum(torch.pairwise_distance(hidden_states, d2)).
 ↪item()) # should be very small (<0.01)
except:
    print("NOT IMPLEMENTED")
```

```
Difference: 0.0017131903441622853
```

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.

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

[5]:
```
model = ClassificationTransformer(word_to_ix, hidden_dim=128, num_heads=2,␣
 ↪dim_feedforward=2048, dim_k=96,
                                  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()) # should be very small (<0.01)
except:
    print("NOT IMPLEMENTED")
```

```
Difference: 0.0017168150516226888
```

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.

[6]:
```
model = ClassificationTransformer(word_to_ix, hidden_dim=128, num_heads=2,␣
 ↪dim_feedforward=2048, dim_k=96,
                                   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())␣
 ↪# should be very small (<1e-5)
except:
    print("NOT IMPLEMENTED")
```

```
Difference: 1.9552965113689424e-06
```

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.

[8]:
```
inputs = train_inxs[0:2]
inputs = torch.LongTensor(inputs)

outputs = model.forward(inputs)

try:
    print("Difference:", torch.sum(torch.pairwise_distance(outputs, scores)).
 ↪item()) # should be very small (<1e-5)
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.

```
# 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.token_embeddings =
nn.Embedding(num_embeddings=self.vocab_size,embedding_dim=self.hidden_dim)
        self.positional_embeddings =
nn.Embedding(num_embeddings=self.max_length,embedding_dim=self.hidden_dim)



################################################################################
        #                       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.ffdLayer1 = nn.Linear(self.hidden_dim,self.dim_feedforward)
    self.reluLayer = nn.ReLU()
    self.ffdLayer2 = 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.finalLayer = nn.Linear(self.hidden_dim,1)
    self.sigmoid = nn.Sigmoid()


##############################################################################
    #                       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.  #

##############################################################################
    embeddings = self.embed(inputs)
    multi_head_attention = self.multi_head_attention(embeddings)
    ffd_layer_output = self.feedforward_layer(multi_head_attention)
    outputs = self.final_layer(ffd_layer_output)


##############################################################################
    #                       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.token_embeddings(inputs)
    embeddings += self.positional_embeddings(torch.arange(inputs.shape[1]))
```

```
##############################################################################
    #                       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.   #

##############################################################################
    # Source: https://stackoverflow.com/questions/50826644/why-do-we-do-batch-matrix-
matrix-product

    #softmax1 = self.softmax(self.q1(inputs).bmm(self.k1(inputs).transpose(1,2)))
    atn1 =
torch.bmm(self.softmax(self.q1(inputs).bmm(self.k1(inputs).transpose(1,2))/np.sqrt(self.dim_k)
),self.v1(inputs))


    #softmax2 = self.softmax(self.q2(inputs).bmm(self.k2(inputs).transpose(1,2)))
    atn2 =
torch.bmm(self.softmax(self.q2(inputs).bmm(self.k2(inputs).transpose(1,2))/np.sqrt(self.dim_k)
),self.v2(inputs))


    outputs = self.attention_head_projection(torch.cat((atn1,atn2), dim=2))

    outputs = self.norm_mh(inputs + outputs)
```

```
##############################################################################
    #                       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.                         #

##############################################################################
    ffd1_o = self.ffdLayer1(inputs)
    relu_o = self.reluLayer(ffd1_o)
    ffd2_o = self.ffdLayer2(relu_o)
    outputs = self.norm_ff(ffd2_o+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.                           #

##############################################################################
    inputs = inputs[:,0,:].squeeze(1)
    final_o = self.finalLayer(inputs)
    outputs = self.sigmoid(final_o)


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