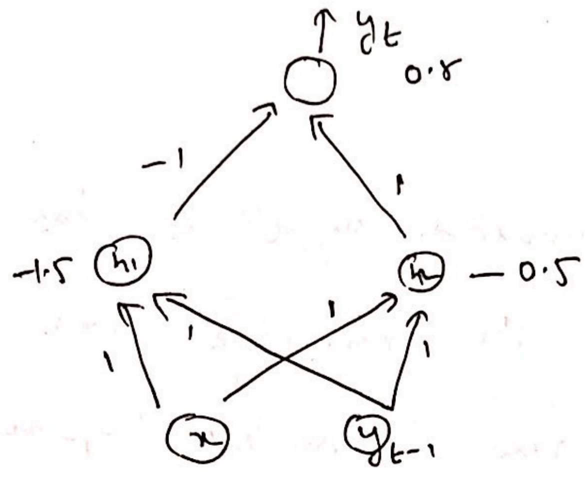
1. Let's use the following RNN with 2 hidden layers and output passing as input in next step.



[Variant of Jordan network].

Stept:

$h_1 = \sigma(x + y_{t-1} - 1.5) \rightarrow$ And gate

$h_2 = \sigma(x + y_{t-1} - 0.5) \rightarrow$ Or gate.

$y_t = \sigma(-h_1 + h_2 - 0.5) = \sigma(-(xy) + (x+y) - 0.5)$

where $\sigma(x) = 1$ if $x > 0$; $y_0 = 0$ for initial step.
0 otherwise.

x_t	y_{t-1}	h_{1t}	h_{2t}	y_t
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

2. The idea is to have $c_t = \bar{x}_t \cdot c_{t-1} + x_t \cdot \overline{c_{t-1}}$,

so that this becomes a XOR function, and hence the output.

Specifics: $c_t = h_t \ \forall t$; $c_0 = h_0 = 0$.

Say at t^{th} step we have $c_{t-1} = h_{t-1}$ and x_t.

$$f_t = \sigma\left([0 \ \ -1]\begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + 0.5\right) = \sigma(-x_t + 0.5) = \bar{x}_t$$

$$i_t = \sigma\left([0 \ \ 1]\begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + 0\right) = \sigma(x_t) = x_t$$

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

$$= \overline{h_{t-1}} = \overline{c_{t-1}}$$

$$o_t = \sigma\left([0 \ \ 0]\begin{bmatrix} h_{t-1} \\ x_t \end{bmatrix} + 1\right) = 1.$$

$$\therefore c_t = \bar{x}_t \cdot c_{t-1} + x_t \cdot \overline{c_{t-1}} = x_t \text{ XOR } c_{t-1}$$

$$h_t = 1 \cdot \tanh(c_t) = c_t \quad [\text{Invariance holds}].$$

Combining all, we get.

$w_f = [0 \ -1], \quad b_f = 0.5$

$w_i = [0 \ 1], \quad b_i = 0$

$w_c = [-1 \ 0], \quad b_c = 1$

$w_o = [0 \ 0], \quad b_o = 1.$

3. Given the best completed hypothesis has score more than all the incomplete ones.

When we complete a new incomplete hypothesis, its score is going to be existing score $\times \Pi P(.)$.

where $\Pi(P(.))$ is product of some probabilities.

As probability is always ≤ 1, the score of a sentence after completion can never be more than existing score before completion. As this is already less than the best hypothesis, score of a newly completed sentence can never be more than 'best' defined in the question.

4. Given $h_t = W^T h_{t-1}$.

$$\frac{\partial L}{\partial h_{t-1}} = W \cdot \frac{\partial L}{\partial h_t}.$$

$$\frac{\partial L}{\partial h_{t-2}} = W^2 \cdot \frac{\partial L}{\partial t}.$$

$$\vdots$$

$$\frac{\partial L}{\partial h_0} = W^{t} \cdot \frac{\partial L}{\partial t}.$$

As W has n eigen values, W can be split as

$W = Q \lambda Q^{-1}$, where λ is diagonal matrix with λ_{ii} = eigenvalue;

$$\therefore \frac{\partial L}{\partial h_0} = Q \lambda^{t} \cdot Q^{-1} \frac{\partial L}{\partial t}.$$

$$= Q \cdot \begin{bmatrix} e_1^{t} & & \\ & e_2^{t} & \\ & & \ddots \end{bmatrix} \frac{\partial L}{\partial t},$$

$e_1, e_2 \ldots$ are eigen values ($\lambda_1, \lambda_2 \ldots$)

Given $\rho(W) = \max(|\lambda_1|, |\lambda_2| \ldots)$.

$\therefore$ As $t \to \infty$, $\lambda^t \to \infty$ if $\lambda > 1$ and

$\lambda^t \to 0$ if $\lambda < 1$.

hence this gets multiplied to derivative and results in vanishing & ~~exp~~ exploding gradient problem.

RNN_Captioning

October 26, 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.

[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_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))))
```

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:

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

```
Requirement already satisfied: h5py in /home/gondi/anaconda3/lib/python3.7/site-
packages (2.9.0)
Requirement already satisfied: numpy>=1.7 in
/home/gondi/anaconda3/lib/python3.7/site-packages (from h5py) (1.16.4)
Requirement already satisfied: six in /home/gondi/anaconda3/lib/python3.7/site-
packages (from h5py) (1.12.0)
```

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:

[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**.

[4]:
```
# 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 man standing next to an elephant with a long trunk <END>

<START> a large machine <UNK> many doughnuts with <UNK> <END>

<START> a <UNK> of wild elephants having <UNK> in a <UNK> hole <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.

[5]:
```
N, D, H = 3, 10, 4

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

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

```
 [ 0.97934501,  0.99144213,  0.99646691,  0.99854353]])

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

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

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.

[6]:
```
from cs231n.rnn_layers import rnn_step_forward, rnn_step_backward
np.random.seed(231)
N, D, H = 4, 5, 6
x = np.random.randn(N, D)
h = np.random.randn(N, H)
Wx = np.random.randn(D, H)
Wh = np.random.randn(H, H)
b = np.random.randn(H)

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

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

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

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

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

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

```
dx error:  4.0192769090159184e-10
dprev_h error:  2.5136656668664053e-10
```

```
dWx error:  3.398875305713782e-10
dWh error:  3.355162782632426e-10
db error:  1.946925061042176e-10
```

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`.

[7]:
```
N, T, D, H = 2, 3, 4, 5

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

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

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

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.

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

N, D, T, H = 2, 3, 10, 5

x = np.random.randn(N, T, D)
h0 = np.random.randn(N, H)
```

```
Wx = np.random.randn(D, H)
Wh = np.random.randn(H, H)
b = np.random.randn(H)

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

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

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

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

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

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

```
dx error:  2.12371994872961e-09
dh0 error:  3.380520197084487e-09
dWx error:  7.133880725895019e-09
dWh error:  1.2991706887909817e-07
db error:  4.309473374164083e-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`.

[9]:
```
N, T, V, D = 2, 4, 5, 3

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

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

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

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

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`.

[10]:
```
np.random.seed(231)

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

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

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

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

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

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.

[11]:
```python
np.random.seed(231)

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

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

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

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

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

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

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

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

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.

[12]:
```
# Sanity check for temporal softmax loss
from cs231n.rnn_layers import temporal_softmax_loss

N, T, V = 100, 1, 10

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

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

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

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

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

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

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

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

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`.

[13]:
```
N, D, W, H = 10, 20, 30, 40
word_to_idx = {'<NULL>': 0, 'cat': 2, 'dog': 3}
V = len(word_to_idx)
```

```
T = 13

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

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

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

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

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

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

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

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

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

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

model = CaptioningRNN(word_to_idx,
          input_dim=input_dim,
          wordvec_dim=wordvec_dim,
          hidden_dim=hidden_dim,
          cell_type='rnn',
```

```
        dtype=np.float64,
    )

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

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

```
W_embed relative error: 2.331074e-09
W_proj relative error: 9.974427e-09
W_vocab relative error: 2.875061e-09
Wh relative error: 4.685196e-09
Wx relative error: 7.725620e-07
b relative error: 4.909225e-10
b_proj relative error: 1.934808e-08
b_vocab relative error: 1.781169e-09
```

13 Overfit small data

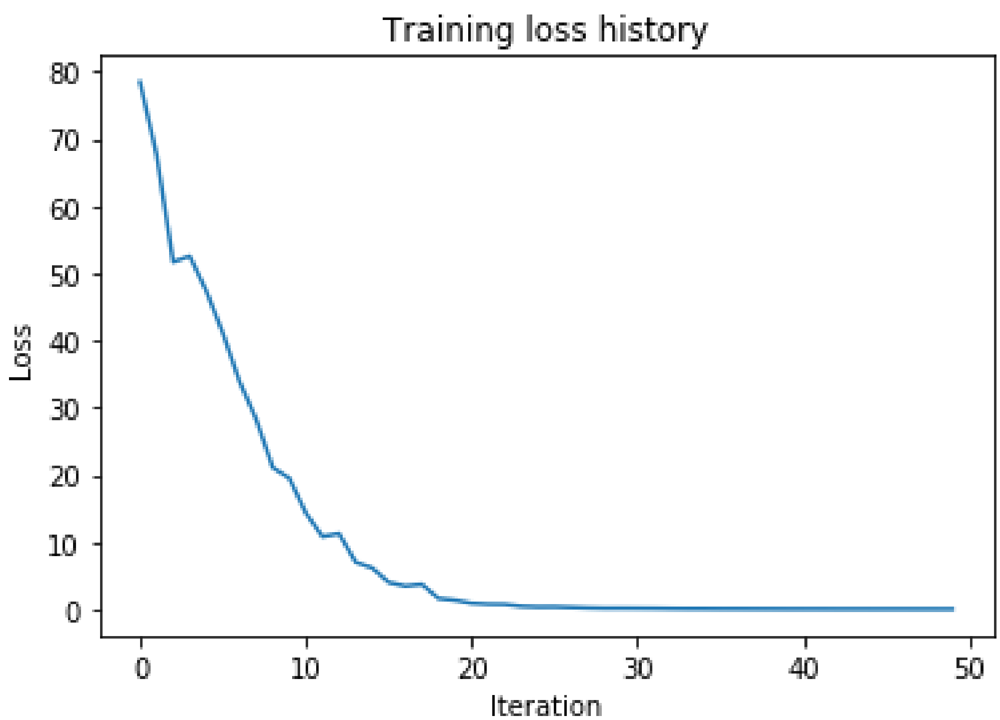
Similar to the `Solver` class that we used to train image classification models on the previous assignment, on this assignment we use a `CaptioningSolver` class to train image captioning models. Open the file `cs231n/captioning_solver.py` and read through the `CaptioningSolver` class; it should look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

[18]:
```
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=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': 0.01,
          },
          lr_decay=0.99,
          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 / 50) loss: 78.491359
(Iteration 11 / 50) loss: 14.336760
(Iteration 21 / 50) loss: 0.905723
(Iteration 31 / 50) loss: 0.158582
(Iteration 41 / 50) loss: 0.043081
```



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.

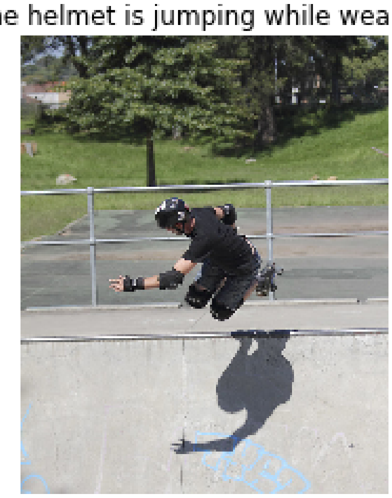
Note: Some of the URLs are missing and will throw an error; re-run this cell until the output is at least 2 good caption samples.

[20]:
```
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
<START> a woman is kneeling near some large <UNK> of food <END>
GT:<START> a woman is kneeling near some large <UNK> of food <END>

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

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



val
<START> <UNK> with the room and <UNK> up in the house <END>
GT:<START> a little boy sitting on the stairs with a racquet <END>

```
[ ]:
```

LSTM_Captioning

October 26, 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.

[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_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))))
```

2 Load MS-COCO data

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

[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.

3.0.1 Removing LSTM description because nbconvert isn't working

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.

[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)
```

```python
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.

[4]:
```python
a = np.zeros((1,5))
b = np.zeros((1,6))
np.concatenate((a,b), axis=1).shape
```

[4]:
```
(1, 11)
```

[5]:
```python
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)
```

```python
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:  6.335032254429549e-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`.

[6]:
```
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.

[7]:
```
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:  7.838503094280515e-09
dh0 error:  2.469092580080717e-08
dWx error:  4.748336603829709e-09
dWh error:  1.0424408314821048e-06
db error:  1.9152724138992987e-09
```

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`.

[8]:
```
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:  9.82445935443226
expected loss:  9.82445935443
difference:  2.261302256556519e-12
```

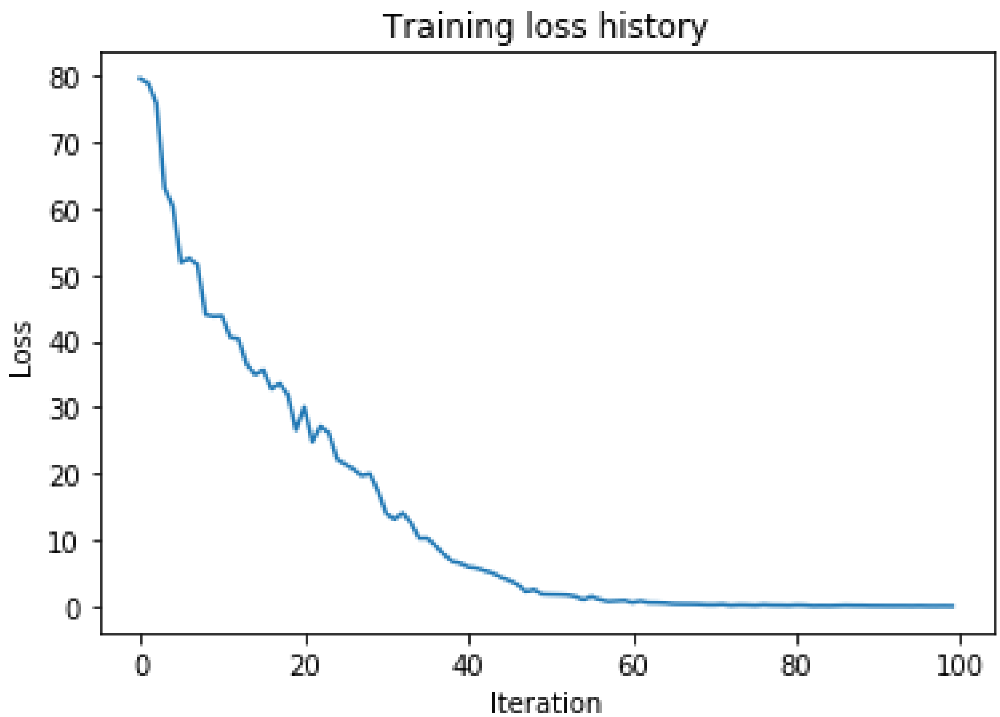
9 Overfit LSTM captioning model

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.

[10]:
```
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: 30.062626
(Iteration 31 / 100) loss: 14.020088
(Iteration 41 / 100) loss: 6.004192
(Iteration 51 / 100) loss: 1.854826
(Iteration 61 / 100) loss: 0.641513
(Iteration 71 / 100) loss: 0.286132
(Iteration 81 / 100) loss: 0.237630
(Iteration 91 / 100) loss: 0.126009
```



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.

[11]:
```
for split in ['train', 'val']:
    minibatch = sample_coco_minibatch(small_data, split=split, batch_size=2)
```

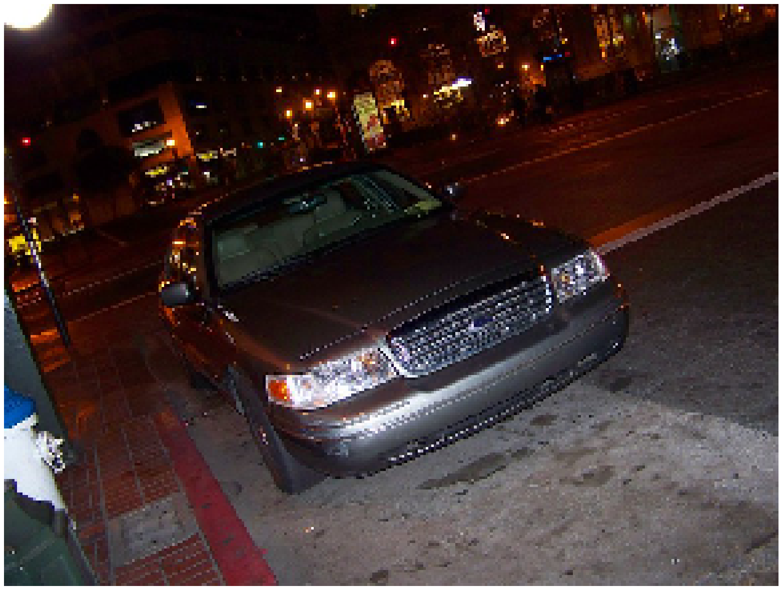
```
    gt_captions, features, urls = minibatch
    gt_captions = decode_captions(gt_captions, data['idx_to_word'])

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

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

val
<START> a cat is <UNK> and a <UNK> <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.

[13]:
```
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]))
```

12 write a description of your model here:

I used Numpy based LSTM model I implementd to get the best model. Because the memory was not sufficient on my laptop, it kept crashing. So I trained it on colab and I'm attaching the colab's output here in this pdf.

12.0.1 Technical details -

Max_train in load_coco_data = 25k Enable_PCA = True Hidden_dim = 512 Wordvec_dim = 256 cell_type = 'lstm'

Number of epochs = 3 + 3 + 3 - Trained 3 epochs each, checked the BLEU scores and repeated this process thrice. Finally got a BLEU score of 0.27

[16]:
```
# 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.
```

[ ]:
```

```

Best_Model

October 26, 2019

[0]:
```
# 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_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))))
```

[0]:
```
enable_PCA = True

# 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=enable_PCA)
```

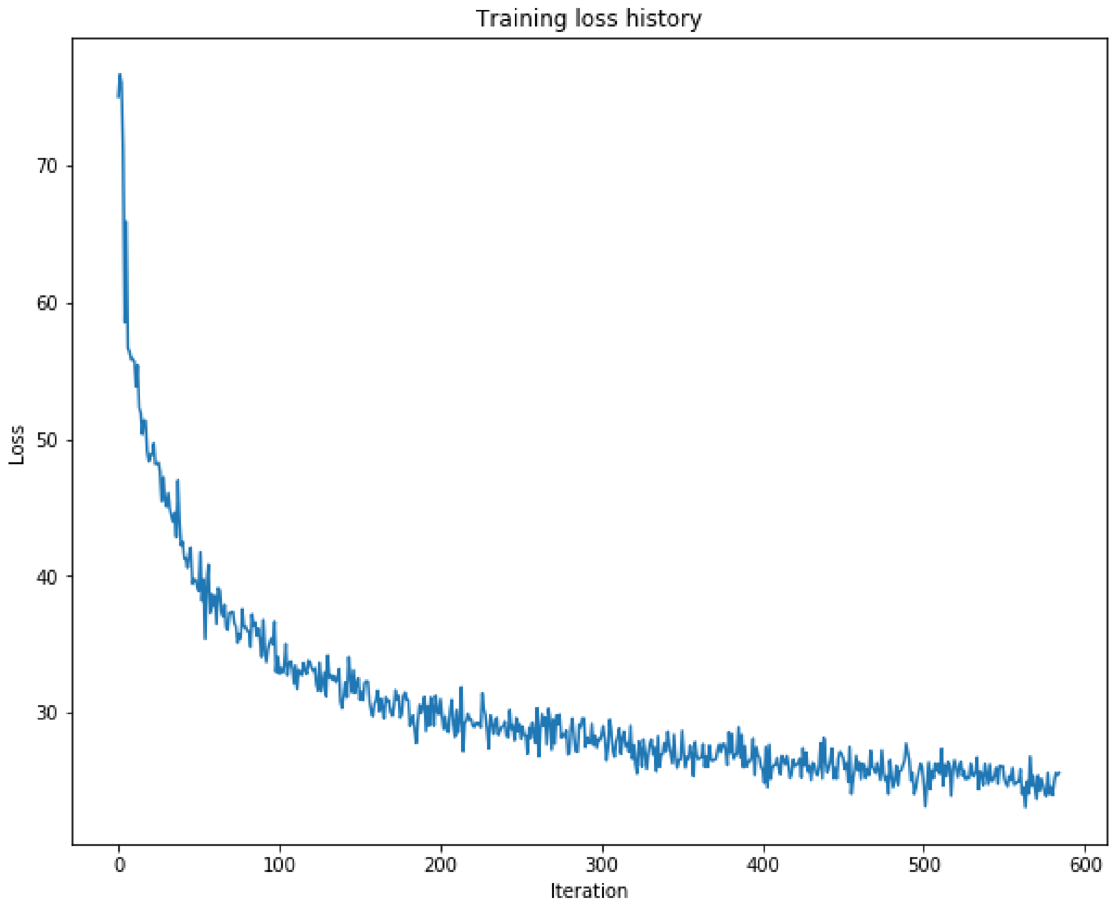
```
# Print out all the keys and values from the data dictionary
for k, v in data.items():
    if type(v) == np.ndarray:
        print(k, type(v), v.shape, v.dtype)
    else:
        print(k, type(v), len(v))
```

[0]:
```
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]))
```

[15]:
```
np.random.seed(231)
my_data = load_coco_data(max_train=25000, pca_features=enable_PCA)

my_lstm_model = CaptioningRNN(
          cell_type='lstm',
          word_to_idx=my_data['word_to_idx'],
          input_dim=my_data['train_features'].shape[1],
          hidden_dim=512,
          wordvec_dim=256,
        )

my_lstm_solver = CaptioningSolver(my_lstm_model, my_data,
           update_rule='adam',
           num_epochs=3,
           batch_size=128,
           optim_config={
             'learning_rate': 5e-3,
           },
           lr_decay=0.95,
           verbose=True, print_every=10,
         )

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()
```

```
(Iteration 1 / 585) loss: 75.026991
(Iteration 11 / 585) loss: 55.676091
(Iteration 21 / 585) loss: 48.848976
(Iteration 31 / 585) loss: 45.058067
(Iteration 41 / 585) loss: 42.505531
(Iteration 51 / 585) loss: 38.904480
(Iteration 61 / 585) loss: 38.542219
(Iteration 71 / 585) loss: 37.298039
(Iteration 81 / 585) loss: 35.966706
(Iteration 91 / 585) loss: 36.765397
(Iteration 101 / 585) loss: 32.830161
(Iteration 111 / 585) loss: 33.430491
(Iteration 121 / 585) loss: 33.234870
(Iteration 131 / 585) loss: 34.193425
(Iteration 141 / 585) loss: 31.274002
(Iteration 151 / 585) loss: 30.930439
```

```
(Iteration 161 / 585) loss: 30.903586
(Iteration 171 / 585) loss: 29.768351
(Iteration 181 / 585) loss: 30.924226
(Iteration 191 / 585) loss: 31.148088
(Iteration 201 / 585) loss: 30.996755
(Iteration 211 / 585) loss: 30.181153
(Iteration 221 / 585) loss: 29.072456
(Iteration 231 / 585) loss: 27.371498
(Iteration 241 / 585) loss: 29.329122
(Iteration 251 / 585) loss: 28.075443
(Iteration 261 / 585) loss: 30.339940
(Iteration 271 / 585) loss: 29.496362
(Iteration 281 / 585) loss: 27.052344
(Iteration 291 / 585) loss: 27.470924
(Iteration 301 / 585) loss: 28.347507
(Iteration 311 / 585) loss: 28.251545
(Iteration 321 / 585) loss: 27.647753
(Iteration 331 / 585) loss: 27.751728
(Iteration 341 / 585) loss: 27.157792
(Iteration 351 / 585) loss: 28.680972
(Iteration 361 / 585) loss: 26.582350
(Iteration 371 / 585) loss: 26.543236
(Iteration 381 / 585) loss: 27.033600
(Iteration 391 / 585) loss: 26.032152
(Iteration 401 / 585) loss: 25.816862
(Iteration 411 / 585) loss: 26.001616
(Iteration 421 / 585) loss: 26.090242
(Iteration 431 / 585) loss: 25.054030
(Iteration 441 / 585) loss: 25.632201
(Iteration 451 / 585) loss: 26.652687
(Iteration 461 / 585) loss: 26.356239
(Iteration 471 / 585) loss: 25.595270
(Iteration 481 / 585) loss: 25.380004
(Iteration 491 / 585) loss: 27.138775
(Iteration 501 / 585) loss: 24.701434
(Iteration 511 / 585) loss: 25.527133
(Iteration 521 / 585) loss: 25.931922
(Iteration 531 / 585) loss: 25.499818
(Iteration 541 / 585) loss: 24.740914
(Iteration 551 / 585) loss: 25.188676
(Iteration 561 / 585) loss: 25.873165
(Iteration 571 / 585) loss: 23.689867
(Iteration 581 / 585) loss: 23.964703
```



[17]:
```
my_lstm_solver.train()
```

```
(Iteration 1 / 585) loss: 25.795926
(Iteration 11 / 585) loss: 23.832439
(Iteration 21 / 585) loss: 24.046382
(Iteration 31 / 585) loss: 23.901144
(Iteration 41 / 585) loss: 23.938254
(Iteration 51 / 585) loss: 23.731246
(Iteration 61 / 585) loss: 23.630522
(Iteration 71 / 585) loss: 23.838357
(Iteration 81 / 585) loss: 23.697402
(Iteration 91 / 585) loss: 23.724039
(Iteration 101 / 585) loss: 22.990490
(Iteration 111 / 585) loss: 24.748578
(Iteration 121 / 585) loss: 23.333434
(Iteration 131 / 585) loss: 25.234987
(Iteration 141 / 585) loss: 23.324452
(Iteration 151 / 585) loss: 25.115228
(Iteration 161 / 585) loss: 23.949237
```

```
(Iteration 171 / 585) loss: 23.255166
(Iteration 181 / 585) loss: 23.090668
(Iteration 191 / 585) loss: 22.991409
(Iteration 201 / 585) loss: 21.457049
(Iteration 211 / 585) loss: 22.512872
(Iteration 221 / 585) loss: 23.251845
(Iteration 231 / 585) loss: 22.065603
(Iteration 241 / 585) loss: 22.298050
(Iteration 251 / 585) loss: 22.684208
(Iteration 261 / 585) loss: 22.308873
(Iteration 271 / 585) loss: 22.487698
(Iteration 281 / 585) loss: 22.624099
(Iteration 291 / 585) loss: 21.964584
(Iteration 301 / 585) loss: 23.393972
(Iteration 311 / 585) loss: 23.426272
(Iteration 321 / 585) loss: 22.426719
(Iteration 331 / 585) loss: 22.115908
(Iteration 341 / 585) loss: 21.954920
(Iteration 351 / 585) loss: 22.406289
(Iteration 361 / 585) loss: 22.186297
(Iteration 371 / 585) loss: 22.712520
(Iteration 381 / 585) loss: 20.691459
(Iteration 391 / 585) loss: 20.868234
(Iteration 401 / 585) loss: 21.194585
(Iteration 411 / 585) loss: 20.931285
(Iteration 421 / 585) loss: 23.286477
(Iteration 431 / 585) loss: 21.412053
(Iteration 441 / 585) loss: 22.689030
(Iteration 451 / 585) loss: 20.910021
(Iteration 461 / 585) loss: 23.465626
(Iteration 471 / 585) loss: 21.728491
(Iteration 481 / 585) loss: 21.361348
(Iteration 491 / 585) loss: 20.692809
(Iteration 501 / 585) loss: 21.341525
(Iteration 511 / 585) loss: 21.129221
(Iteration 521 / 585) loss: 21.053841
(Iteration 531 / 585) loss: 20.129097
(Iteration 541 / 585) loss: 21.371359
(Iteration 551 / 585) loss: 21.567446
(Iteration 561 / 585) loss: 20.704495
(Iteration 571 / 585) loss: 21.335061
(Iteration 581 / 585) loss: 20.070246
```

[19]:
```
my_lstm_solver.train()
evaluate_model(my_lstm_model)
```

```
(Iteration 1 / 585) loss: 20.381503
(Iteration 11 / 585) loss: 20.448205
```

```
(Iteration 21 / 585) loss: 21.324784
(Iteration 31 / 585) loss: 21.609079
(Iteration 41 / 585) loss: 21.580309
(Iteration 51 / 585) loss: 20.903560
(Iteration 61 / 585) loss: 20.007571
(Iteration 71 / 585) loss: 20.725409
(Iteration 81 / 585) loss: 18.594530
(Iteration 91 / 585) loss: 20.317822
(Iteration 101 / 585) loss: 19.378359
(Iteration 111 / 585) loss: 20.556345
(Iteration 121 / 585) loss: 20.003218
(Iteration 131 / 585) loss: 20.837640
(Iteration 141 / 585) loss: 20.797092
(Iteration 151 / 585) loss: 20.237911
(Iteration 161 / 585) loss: 19.655144
(Iteration 171 / 585) loss: 20.924835
(Iteration 181 / 585) loss: 18.762693
(Iteration 191 / 585) loss: 20.585936
(Iteration 201 / 585) loss: 19.442318
(Iteration 211 / 585) loss: 18.371262
(Iteration 221 / 585) loss: 19.566462
(Iteration 231 / 585) loss: 20.415031
(Iteration 241 / 585) loss: 20.370510
(Iteration 251 / 585) loss: 19.418221
(Iteration 261 / 585) loss: 18.359681
(Iteration 271 / 585) loss: 17.743266
(Iteration 281 / 585) loss: 19.034277
(Iteration 291 / 585) loss: 20.286607
(Iteration 301 / 585) loss: 19.186652
(Iteration 311 / 585) loss: 19.540351
(Iteration 321 / 585) loss: 18.852276
(Iteration 331 / 585) loss: 19.013929
(Iteration 341 / 585) loss: 18.663316
(Iteration 351 / 585) loss: 19.164750
(Iteration 361 / 585) loss: 19.294628
(Iteration 371 / 585) loss: 19.664992
(Iteration 381 / 585) loss: 18.588801
(Iteration 391 / 585) loss: 17.874636
(Iteration 401 / 585) loss: 18.827716
(Iteration 411 / 585) loss: 18.577948
(Iteration 421 / 585) loss: 17.897742
(Iteration 431 / 585) loss: 18.114163
(Iteration 441 / 585) loss: 18.978039
(Iteration 451 / 585) loss: 17.997055
(Iteration 461 / 585) loss: 17.920368
(Iteration 471 / 585) loss: 17.319174
(Iteration 481 / 585) loss: 17.867872
(Iteration 491 / 585) loss: 16.459004
```

```
(Iteration 501 / 585) loss: 17.449187
(Iteration 511 / 585) loss: 18.363160
(Iteration 521 / 585) loss: 18.920420
(Iteration 531 / 585) loss: 17.658245
(Iteration 541 / 585) loss: 16.796391
(Iteration 551 / 585) loss: 17.688266
(Iteration 561 / 585) loss: 18.113339
(Iteration 571 / 585) loss: 17.276459
(Iteration 581 / 585) loss: 17.747088
Average BLEU score for train: 0.266838
Average BLEU score for val: 0.270120
```

Transformer_Classification

October 26, 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.

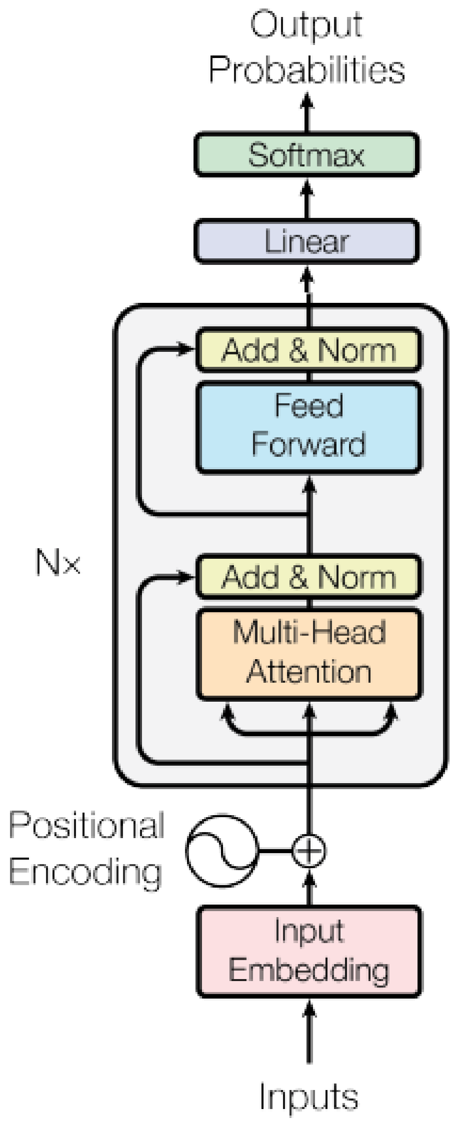
We've additionally already constructed a vocabulary and converted all of the strings of tokens into integers which can be used for vocabulary lookup for you. Feel free to explore the data here.

[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:



imgs/encoder.png

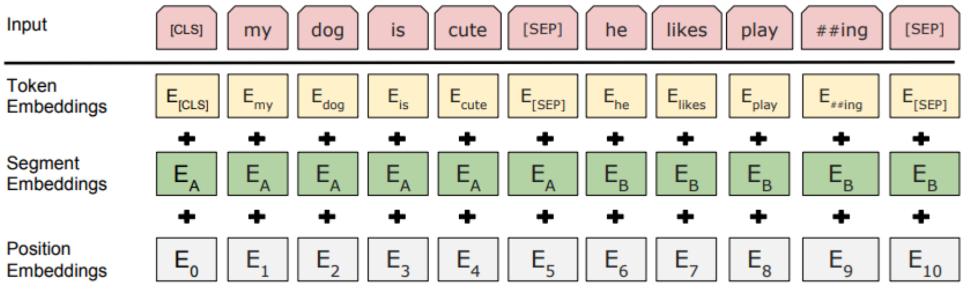
You can refer to the original paper for more details.

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: 0.0017998494440689683
```

1.5 Deliverable 2: Multi-head Self-Attention

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

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

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.

[5]:
```
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")
```

```
torch.Size([2, 43, 128])
Difference: 0.0017151650972664356
```

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.

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

imgs/ffn.png

[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])

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: 0.0017135577509179711
```

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.

[7]:
```
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")
```

```
torch.Size([2, 128])
Difference: 1.8956918665935518e-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")
```

```
torch.Size([2, 43, 128])
torch.Size([2, 128])
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_embedding = nn.Embedding(num_embeddings = self.vocab_size, embedding_dim =
self.hidden_dim)
        self.positional_encoding = nn.Embedding(num_embeddings = self.max_length, embedding_di
m = 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_di
m)
        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.feedForwardLayer1 = nn.Linear(self.hidden_dim, self.dim_feedforward)
        self.relu_layer = nn.ReLU()
        self.feedForwardLayer2 = 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.final_linear_layer = 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.  #
        #############################################################################

        outputs = self.embed(inputs)
```

```
        outputs = self.multi_head_attention(outputs)
        outputs = self.feedforward_layer(outputs)
        outputs = self.final_layer(outputs)

        ##############################################################################
        #                               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)
        """

        #############################################################################
        # 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.                                               #
        #############################################################################
        N, T = inputs.shape
        embeddings = self.token_embedding(inputs)
        embeddings += self.positional_encoding(torch.arange(T))

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

        attention1 = torch.bmm(self.softmax(self.q1(inputs).bmm(self.k1(inputs).transpose(1,2)
) / np.sqrt(self.dim_k)), self.v1(inputs))
        attention2 = 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((attention1, attention2), dim = 2))
        outputs = self.norm_mh(inputs + outputs)
        print(outputs.shape)
        ##############################################################################
        #                               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.feedForwardLayer1(inputs)
        outputs = self.relu_layer(outputs)
        outputs = self.feedForwardLayer2(outputs)
        outputs = self.norm_ff(outputs + 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.                        #
        ############################################################################

        final_tokens = inputs[:, 0, :].squeeze(1) # Collecting only CLS outputs
        print(final_tokens.shape)
        outputs = self.final_linear_layer(final_tokens)
        outputs = self.sigmoid(outputs)

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