RNN_Captioning

November 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.

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

from ie590.gradient_check import eval_numerical_gradient,␣
 ↪eval_numerical_gradient_array
from ie590.rnn_layers import *
from ie590.captioning_solver import CaptioningSolver
from ie590.classifiers.rnn import CaptioningRNN
from ie590.coco_utils import load_coco_data, sample_coco_minibatch,␣
 ↪decode_captions
from ie590.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 /opt/anaconda3/lib/python3.7/site-
packages (2.9.0)
Requirement already satisfied: numpy>=1.7 in /opt/anaconda3/lib/python3.7/site-
packages (from h5py) (1.16.4)
Requirement already satisfied: six in /opt/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 `ie590/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 `ie590/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 `ie590/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 `ie590/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 brown cow walks on a road with two motor bikes in the background <END>

<START> an smoke trail behind an airplane flying in the air <END>

<START> a round cheese pizza with toppings on a white plate <END>

3 Recurrent Neural Networks

As discussed in lecture, we will use recurrent neural network (RNN) language models for image captioning. The file `ie590/rnn_layers.py` contains implementations of different layer types that are needed for recurrent neural networks, and the file `ie590/classifiers/rnn.py` uses these layers to implement an image captioning model.

We will first implement different types of RNN layers in `ie590/rnn_layers.py`.

4 Vanilla RNN: step forward

Open the file `ie590/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. It is common to use tanh in the RNN cell. However, in this homework, you are asked to implement the RNN cell that takes as an argument 'use_tanh'. If use_tanh is True, use np.tanh function, else, use sigmoid function defined in ie590/rnn_layers.py. After doing so run the following to check your implementation. You should see errors on the order of e-8 or less.

[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)

############################
### Use tanh in RNN Cell ###
############################
USE_TANH = True
next_h, _ = rnn_step_forward(x, prev_h, Wx, Wh, b, use_tanh = USE_TANH)
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))

##############################
### Use sigmoid in RNN Cell ###
##############################
## Note that use_tanh flag is set to False
next_h, _ = rnn_step_forward(x, prev_h, Wx, Wh, b, use_tanh = False)
expected_next_h = np.asarray([
    [0.3396039,  0.36546169, 0.3921193,  0.41943607],
    [0.69170684, 0.74773577, 0.79657666, 0.83801004],
    [0.90731492, 0.93847897, 0.95963077, 0.97371398]
    ])
print('next_h error: ', rel_error(expected_next_h, next_h))
```

```
next_h error:  6.292421426471037e-09
next_h error:  4.820744797197939e-09
```

5 Vanilla RNN: step backward

In the file `ie590/rnn_layers.py` implement the `rnn_step_backward` function. After doing so run the following to numerically gradient check your implementation. You should see errors on the order of `e-8` or less.

[6]:
```
from ie590.rnn_layers import rnn_step_forward, rnn_step_backward
USE_TANH = True ## Switch it to False and see if your implementation of sigmoid␣
 ↪is correct.

np.random.seed(590)
N, D, H = 5, 6, 7
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, use_tanh = USE_TANH)
```

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

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

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

```
dx error:  3.24765869266674e-10
dprev_h error:  3.682334227081318e-10
dWx error:  1.3975002321627148e-08
dWh error:  2.0387146320530324e-10
db error:  5.687650384131226e-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 processes an entire sequence of data.

In the file `ie590/rnn_layers.py`, implement the function `rnn_forward`. This should be implemented using the `rnn_step_forward` function that you defined above. After doing so run the following to check your implementation. You should see errors on the order of `e-7` or less.

[7]:
```
N, T, D, H = 2, 4, 5, 6
USE_TANH = True

x = np.linspace(-0.2, 0.1, num=N*T*D).reshape(N, T, D)
h0 = np.linspace(-0.2, 0.2, num=N*H).reshape(N, H)
Wx = np.linspace(-0.3, 0.5, num=D*H).reshape(D, H)
Wh = np.linspace(-0.4, 0.3, num=H*H).reshape(H, H)
```

```
b = np.linspace(-0.5, 0.4, num=H)

## By default, we are usign tanh function in the RNN cell.
h, _ = rnn_forward(x, h0, Wx, Wh, b, use_tanh = USE_TANH)
expected_h = np.asarray([[
    [-0.35757005, -0.22854339, -0.09095507,  0.05019554,  0.18936904,    0.
 ↪32132822],
    [-0.20789153, -0.05339845,  0.10369398,  0.25577779,  0.39617425,    0.
 ↪52020447],
    [-0.28852623, -0.11106078,  0.07378196,  0.25369063,  0.41760738,    0.
 ↪5582089 ],
    [-0.22451474, -0.03985937,  0.14755928,  0.32494804,  0.48208299,    0.
 ↪61331587]
  ],
  [
    [-0.44715081, -0.28423792, -0.10307224,  0.08520022,  0.26757888,    0.
 ↪43260827],
    [-0.10621368,  0.07333185,  0.24824648,  0.40834448,  0.54672801,    0.
 ↪66054612],
    [-0.32601186, -0.11486438,  0.10719865,  0.31905786,  0.50320119,    0.
 ↪65073314],
    [-0.1694327,   0.04334766,  0.25226682,  0.44004741,  0.59588814,    0.
 ↪71688379]
  ]])
print('h error: ', rel_error(expected_h, h))
```

```
h error:  4.641478091199054e-08
```

7 Vanilla RNN: backward

In the file `ie590/rnn_layers.py`, implement the backward pass for a vanilla RNN in the function `rnn_backward`. This should run back-propagation over the entire sequence, making calls to the `rnn_step_backward` function that you defined earlier. You should see errors on the order of e-8 or less.

[8]:
```
np.random.seed(590)

N, D, T, H = 2, 4, 12, 6
USE_TANH = True ## Switch it to False and see if your implementation of sigmoid␣
 ↪is correct.

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, use_tanh = USE_TANH)

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

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

fx = lambda x: rnn_forward(x, h0, Wx, Wh, b, use_tanh = USE_TANH)[0]
fh0 = lambda h0: rnn_forward(x, h0, Wx, Wh, b, use_tanh = USE_TANH)[0]
fWx = lambda Wx: rnn_forward(x, h0, Wx, Wh, b, use_tanh = USE_TANH)[0]
fWh = lambda Wh: rnn_forward(x, h0, Wx, Wh, b, use_tanh = USE_TANH)[0]
fb = lambda b: rnn_forward(x, h0, Wx, Wh, b, use_tanh = USE_TANH)[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:  3.6243372482664874e-09
dh0 error:  6.71451402545458e-10
dWx error:  4.430852688705947e-08
dWh error:  1.921472564103353e-08
db error:  3.978609538119464e-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 `ie590/rnn_layers.py`, implement the function `word_embedding_forward` to convert words (represented by integers) into vectors. Run the following to check your implementation. You should see an error on the order of `e-7` or less.

[9]:
```
N, T, V, D = 2, 6, 6, 4

x = np.asarray([[0, 3, 1, 4, 2, 0], [2, 4, 1, 0, 3, 4]])
W = np.linspace(0, 1, num=V*D).reshape(V, D)

out, _ = word_embedding_forward(x, W)
```

```
expected_out = np.asarray([[
  [0.        , 0.04347826, 0.08695652, 0.13043478],
  [0.52173913, 0.56521739, 0.60869565, 0.65217391],
  [0.17391304, 0.2173913,  0.26086957, 0.30434783],
  [0.69565217, 0.73913043, 0.7826087,  0.82608696],
  [0.34782609, 0.39130435, 0.43478261, 0.47826087],
  [0.        , 0.04347826, 0.08695652, 0.13043478]
  ],
 [[0.34782609, 0.39130435, 0.43478261, 0.47826087],
  [0.69565217, 0.73913043, 0.7826087,  0.82608696],
  [0.17391304, 0.2173913,  0.26086957, 0.30434783],
  [0.,         0.04347826, 0.08695652, 0.13043478],
  [0.52173913, 0.56521739, 0.60869565, 0.65217391],
  [0.69565217, 0.73913043, 0.7826087,  0.82608696]]
])

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

```
out error:  1.0000000112083678e-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 an error on the order of `e-11` or less.

[10]:
```
np.random.seed(590)

N, T, V, D = 40, 4, 6, 8
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:  3.2756739469970712e-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 `ie590/rnn_layers.py`. Run the following to perform numeric gradient checking on the implementation. You should see errors on the order of e-9 or less.

[11]:
```
np.random.seed(590)

# Gradient check for temporal affine layer
N, T, D, M = 2, 4, 5, 6
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:  7.219015984629606e-11
dw error:  2.2265994302981633e-11
db error:  9.291618906409797e-12
```

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 `ie590/rnn_layers.py`.

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

[12]:
```
# Sanity check for temporal softmax loss
from ie590.rnn_layers import temporal_softmax_loss

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 = 80, 1, 12
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))
```

```
2.3023015400095255
23.026219922871423
2.3066946397001176
dx error:  1.346611884773111e-07
```

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 `ie590/classifiers/rnn.py` and look at the `CaptioningRNN` class.

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

[13]:
```
N, D, W, H = 16, 18, 36, 48
word_to_idx = {'<NULL>': 0, 'cat': 2, 'dog': 3}
V = len(word_to_idx)
T = 12

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 = 9.69671544894

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

```
loss:  9.696715448942058
expected loss:  9.69671544894
difference:  2.0587975768648903e-12
```

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

[14]:
```
np.random.seed(590)

batch_size = 3
timesteps = 4
input_dim = 4
wordvec_dim = 5
hidden_dim = 8
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: 4.477461e-09
W_proj relative error: 3.799669e-08
W_vocab relative error: 4.674965e-09
Wh relative error: 7.767722e-07
Wx relative error: 1.152356e-06
b relative error: 1.795807e-09
b_proj relative error: 4.054966e-09
b_vocab relative error: 4.753671e-09
```

13 Overfit small data

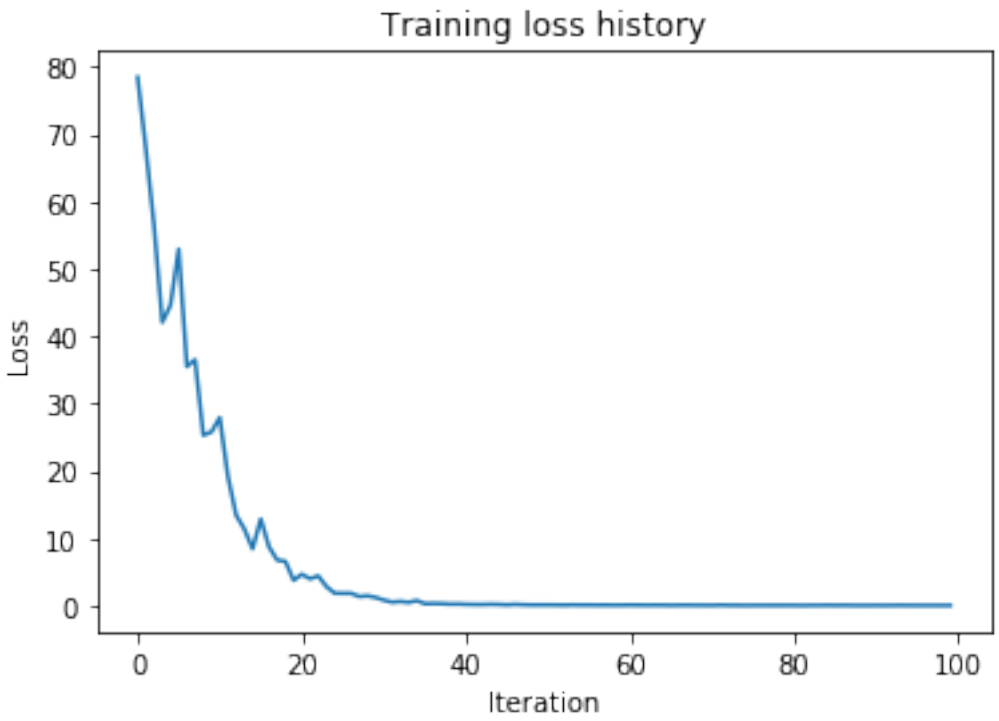
Similar to the `Solver` class that we used to train image classification models on the previous assignment, on this assignment we use a `CaptioningSolver` class to train image captioning models. Open the file `ie590/captioning_solver.py` and read through the `CaptioningSolver` class; it should look very familiar.

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

[15]:
```
np.random.seed(590)

small_data = load_coco_data(max_train=50)

small_rnn_model = CaptioningRNN(
          cell_type='rnn',
          word_to_idx=data['word_to_idx'],
          input_dim=data['train_features'].shape[1],
          hidden_dim=512,
          wordvec_dim=256,
          use_tanh = True
        ) ## Change use_tanh flag to False if you want to use sigmoid in the RNN␣
 ↪cell.
```

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

small_rnn_solver.train()

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

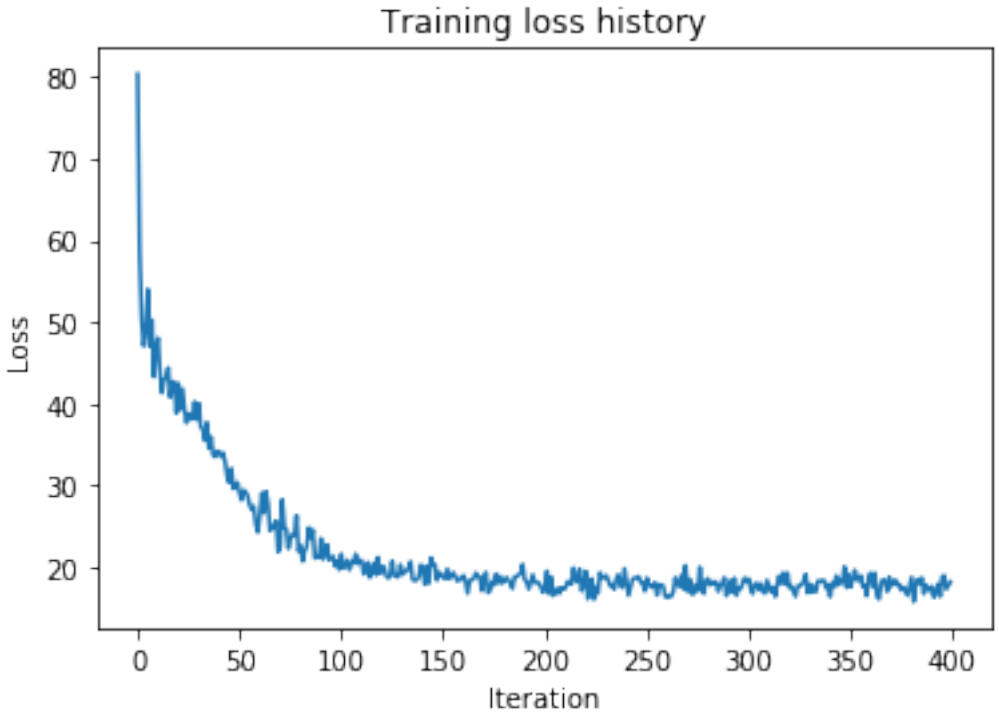
```
(Iteration 1 / 100) loss: 78.465702
(Iteration 11 / 100) loss: 27.946558
(Iteration 21 / 100) loss: 4.701123
(Iteration 31 / 100) loss: 0.858960
(Iteration 41 / 100) loss: 0.236885
(Iteration 51 / 100) loss: 0.153174
(Iteration 61 / 100) loss: 0.136709
(Iteration 71 / 100) loss: 0.086158
(Iteration 81 / 100) loss: 0.085482
(Iteration 91 / 100) loss: 0.084402
```



[19]:
```
np.random.seed(590)

small_data = load_coco_data(max_train=50)

small_rnn_model = CaptioningRNN(
          cell_type='rnn',
          word_to_idx=data['word_to_idx'],
          input_dim=data['train_features'].shape[1],
          hidden_dim=512,
          wordvec_dim=256,
          use_tanh = False
        ) ## Change use_tanh flag to False if you want to use sigmoid in the RNN␣
 ↪cell.

small_rnn_solver = CaptioningSolver(small_rnn_model, small_data,
           update_rule='adam',
           num_epochs=200,
           batch_size=25,
           optim_config={
             'learning_rate': 5e-3,
           },
           lr_decay=0.95,
           verbose=True, print_every=40,
```

```
        )

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 / 400) loss: 80.364909
(Iteration 41 / 400) loss: 34.119761
(Iteration 81 / 400) loss: 22.637750
(Iteration 121 / 400) loss: 20.064523
(Iteration 161 / 400) loss: 18.935156
(Iteration 201 / 400) loss: 17.668102
(Iteration 241 / 400) loss: 18.117628
(Iteration 281 / 400) loss: 18.568855
(Iteration 321 / 400) loss: 17.424112
(Iteration 361 / 400) loss: 19.224539
```



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 `ie590/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.

[16]:
```
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 <UNK> flying through a cloudy gray sky <END>
GT:<START> a <UNK> flying through a cloudy gray sky <END>

val
a large suitcase is on a some cake <END>
GT:<START> a person in <UNK> and red <UNK> stands next to a toilet <END>

val
a <UNK> with many cows lying in the grass <END>
GT:<START> people on an ocean beach on a clear day flying kites <END>

15 INLINE QUESTION 1

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

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

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

Your Answer:

Advantage: The image captioning model with a character level RNN has a very small vocabulary which is in the order of number of characters in English language. Even when there are a million different words, the size of the vocabulary remains the same. Hence we will save a lot in memory that is occupied the embeddings.

Disadvantage: Since we are using characters, the number of timesteps would increase proportionally as there are larger sequences. In the above example, a word level RNN will have 5 hidden layers and a character level RNN will have ten hidden layers. Since there are less iterations per instance, the chances of vanishing gradients and overfitting are lower in the case of word-level RNN.

16 INLINE QUESTION 2

What are the number of parameters and computations in the forward pass of a Vanilla RNN cell? In other words, how many parameters are there at each timestep? Refer to rnn_step_forward function in ie590/rnn_layers.py.

Your Answer:

Parameters are in Wx, Wh, and b. So, the total number is D * H + H * H + H = H*(D+H+1).

17 INLINE QUESTION 3

What is the memory usage in the forward and backward pass of an Vanilla RNN cell? Refer to rnn_step_forward and rnn_step_backward functions in ie590/rnn_layers.py

Your Answer:

Vanilla RNN is more similar to FC layer rather than Conv layer, thus, the parameters also much more and take farily large amount of memory. So, considering memory usage in RNN, I would sum up the input data x, hiden variable h, and parameters (Wx, Wh, b).

The memory usage in forward pass (in unit of bytes per number) is N * D + N * H + H * (D+H+1) = N * D + H * (N+D+H+1). The backward pass takes the same amount of memory as forward pass, because x, h, and parameters have a gradient of the same size as itself.

18 INLINE QUESTION 4

In this question, you have implemented generic functions for rnn_step_forward, rnn_forward, rnn_step_backward and rnn_backward. Try training with RNN cell with tanh and sigmoid. Comment on the observed differences. Specifically, how many iterations are needed for the model to converge (to get an error < 1.0) in both the cases.

Your Answer:

Look at the section "Overfit small data". RNN with tanh can easily converge within 30 iterations at loss < 0.1. But, the RNN with sigmoid takes about 120 iterations to converge at a much larger loss ~ 18. So, the example shows that tanh converges much faster and more accurate than sigmoid, because tanh is zero-centered while sigmoid is always >0. Since sigmoid value is always positive, the backward pass tends to propagate towards one direction, making it harder to find the minimal loss.