LSTM_Captioning

November 25, 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
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))))
```

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 vanilla RNN called Long-Short Term Memory (LSTM) RNNs. Vanilla RNNs can be tough to train on long sequences due to vanishing and exploding gradients caused by repeated matrix multiplication. LSTMs solve this problem by replacing the simple update rule of the vanilla RNN with a gating mechanism as follows.

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

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

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

where σ is the sigmoid function and tanh is the hyperbolic tangent, both applied elementwise.

Finally we compute the next cell state c_t and next hidden state h_t as

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

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

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

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

4 LSTM: step forward

Implement the forward pass for a single timestep of an LSTM in the `lstm_step_forward` function in the file `ie590/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 on the order of `e-7` or less.

[3]:
```
N, D, H = 4, 6, 5
x = np.linspace(-0.3, 1.5, num=N*D).reshape(N, D)
prev_h = np.linspace(-0.4, 0.6, num=N*H).reshape(N, H)
prev_c = np.linspace(-0.3, 0.7, num=N*H).reshape(N, H)
Wx = np.linspace(-2.5, 1.5, num=4*D*H).reshape(D, 4 * H)
Wh = np.linspace(-0.6, 2.9, num=4*H*H).reshape(H, 4 * H)
b = np.linspace(0.5, 0.9, num=4*H)

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

expected_next_h = np.asarray([
    [0.0165847,  0.01696218, 0.01655814, 0.01553876, 0.01406699],
    [0.24065888, 0.28427873, 0.32864975, 0.3732101,  0.41741899],
    [0.20770108, 0.27550721, 0.35119713, 0.43137313, 0.51193919],
    [0.16696798, 0.25014776, 0.35113955, 0.46255914, 0.57402168]
])

expected_next_c = np.asarray([
    [0.02582877, 0.02692254, 0.02680088, 0.0256642,  0.02372262],
    [0.36907851, 0.4306525,  0.49380227, 0.55837699, 0.62422696],
    [0.29771495, 0.37848565, 0.46994136, 0.57115168, 0.68058741],
    [0.22573283, 0.31721055, 0.43100444, 0.56617513, 0.71982552]
])

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:  1.294307513499059e-07
next_c error:  5.6488549743951826e-08
```

5 LSTM: step backward

Implement the backward pass for a single LSTM timestep in the function `lstm_step_backward` in the file `ie590/rnn_layers.py`. Once you are done, run the following to perform numeric gradient checking on your implementation. You should see errors on the order of `e-7` or less.

[4]:
```
np.random.seed(590)

N, D, H = 4, 6, 8
x = np.random.randn(N, D)
prev_h = np.random.randn(N, H)
prev_c = np.random.randn(N, H)
Wx = np.random.randn(D, 4 * H)
Wh = np.random.randn(H, 4 * H)
b = np.random.randn(4 * H)

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

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

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

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

num_grad = eval_numerical_gradient_array

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

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

print('dx error: ', rel_error(dx_num, dx))
print('dh error: ', rel_error(dh_num, dh))
print('dc error: ', rel_error(dc_num, dc))
print('dWx error: ', rel_error(dWx_num, dWx))
print('dWh error: ', rel_error(dWh_num, dWh))
print('db error: ', rel_error(db_num, db))
```

```
dx error:  2.311685145933396e-10
dh error:  1.9010102808883855e-10
dc error:  7.035105968703615e-11
dWx error:  1.0355549616510445e-07
dWh error:  3.365903077111766e-08
db error:  1.8493351481023468e-09
```

6 LSTM: forward

In the function `lstm_forward` in the file `ie590/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 on the order of `e-7` or less.

[5]:
```
N, D, H, T = 2, 6, 5, 4
x = np.linspace(-0.2, 0.5, num=N*T*D).reshape(N, T, D)
h0 = np.linspace(-0.3, 0.9, num=N*H).reshape(N, H)
Wx = np.linspace(-0.1, 0.9, num=4*D*H).reshape(D, 4 * H)
Wh = np.linspace(-0.4, 0.7, num=4*H*H).reshape(H, 4 * H)
b = np.linspace(0.3, 0.5, num=4*H)

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

expected_h = np.asarray([
    [[0.10504642, 0.10522426, 0.10540215, 0.10558009, 0.10575809],
    [0.18395915, 0.18913038, 0.19432382, 0.19953735, 0.20476888],
    [0.34981305, 0.36088469, 0.37188293, 0.38279899, 0.39362474],
    [0.54749071, 0.5627889,  0.57766486, 0.59211578, 0.60614103]],
    [[0.52886909, 0.53975977, 0.55021991, 0.56025857, 0.56988555],
    [0.71124032, 0.72440089, 0.7369036,  0.74877688, 0.76004879],
    [0.83180499, 0.84312229, 0.85362763, 0.86338152, 0.87244016],
    [0.88993707, 0.89870573, 0.90672769, 0.91407466, 0.92081035]]
])

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

```
h error:  2.1438206562814823e-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 `ie590/rnn_layers.py`. When you are done, run the following to perform numeric gradient checking on your implementation. You should see errors on the order of `e-8` or less. (For `dWh`, it's fine if your error is on the order of `e-6` or less).

[6]:
```
from ie590.rnn_layers import lstm_forward, lstm_backward
np.random.seed(590)

N, D, T, H = 2, 6, 12, 8

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:  2.6953417373424663e-08
dh0 error:  8.026997537952733e-07
dWx error:  3.4463537844320296e-08
dWh error:  1.2125091952001396e-07
db error:  2.063275244432631e-08
```

8 INLINE QUESTION 1

Recall that in an LSTM the input gate i, forget gate f, and output gate o are all outputs of a sigmoid function. Why don't we use the ReLU activation function instead of sigmoid to compute these values? Explain.

Your Answer:

Because these gates mean some kind of probabilities, such as forget gate f tells how much to forget from previous time step. For probility, it is better to confine the value between 0 ~ 1. Another reason might be the analog to gates in circuits, whose output is either 0 or 1. Since sigmoid function can make sure the output be 0 ~ 1 whereas ReLU can goes up to infinity, sigmoid makes more sense here.

9 LSTM captioning model

Now that you have implemented an LSTM, update the implementation of the `loss` method of the `CaptioningRNN` class in the file `ie590/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 on the order of `e-10` or less.

[7]:
```
N, D, W, H = 20, 18, 30, 48
word_to_idx = {'<NULL>': 0, 'cat': 2, 'dog': 3}
V = len(word_to_idx)
T = 16

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 = 12.304967965

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

```
loss:  12.30496796520032
expected loss:  12.304967965
difference:  2.0031976077916624e-10
```

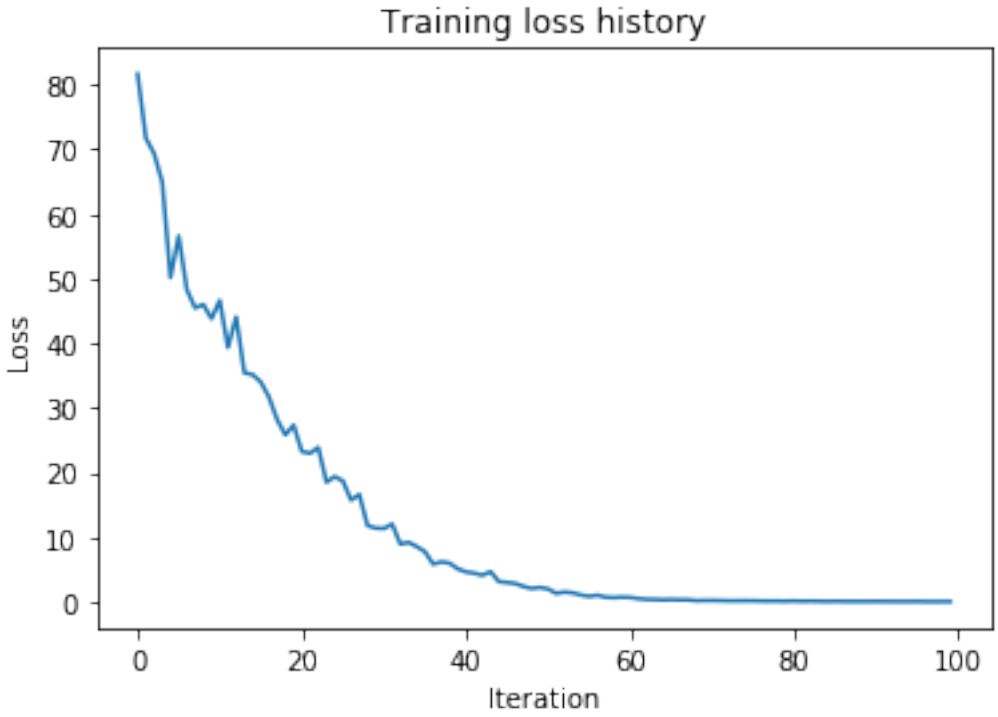
10 Overfit LSTM captioning model

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

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

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: 81.661962
(Iteration 11 / 100) loss: 46.648882
(Iteration 21 / 100) loss: 23.297663
(Iteration 31 / 100) loss: 11.390962
(Iteration 41 / 100) loss: 4.674460
```

```
(Iteration 51 / 100) loss: 2.031479
(Iteration 61 / 100) loss: 0.722321
(Iteration 71 / 100) loss: 0.257914
(Iteration 81 / 100) loss: 0.176969
(Iteration 91 / 100) loss: 0.121069
```



11 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.

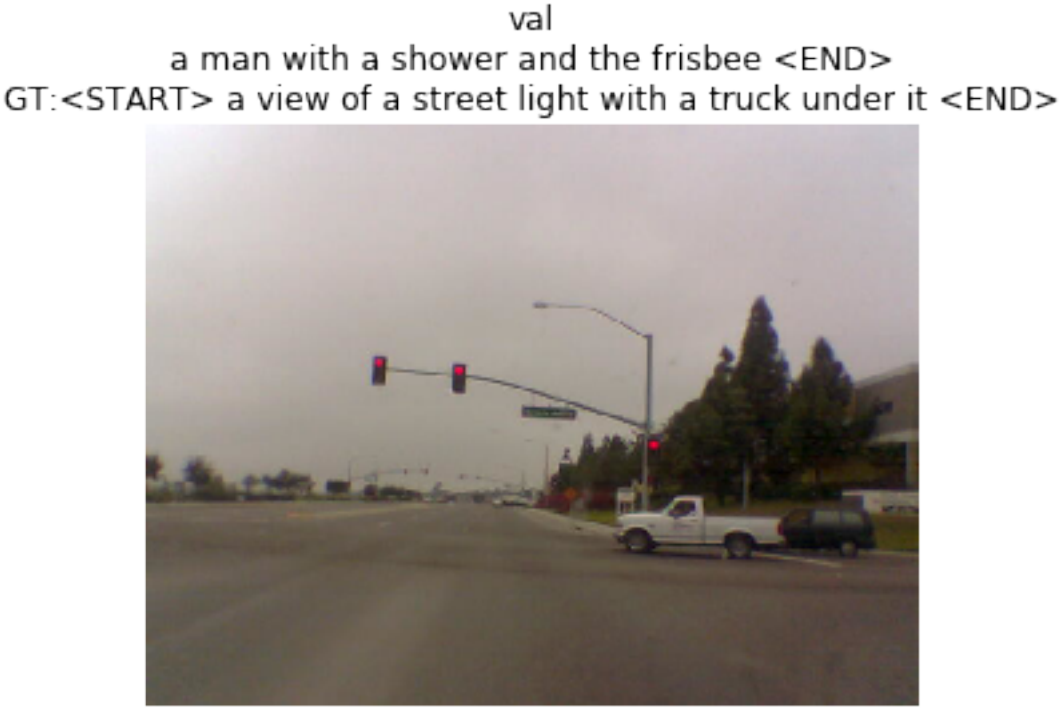
When you are done run the following to sample from your overfit LSTM model on some training and validation set samples. As with the RNN, training results should be very good, and validation results probably won't make a lot of sense (because we're overfitting).

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

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

```
   for gt_caption, sample_caption, url in zip(gt_captions, sample_captions,␣
↪urls):
       plt.imshow(image_from_url(url))
       plt.title('%s\n%s\nGT:%s' % (split, sample_caption, gt_caption))
       plt.axis('off')
       plt.show()
```

train
a guy with a surfboard on a beach <END>
GT:<START> a guy with a surfboard on a beach <END>

val
a <UNK> and <UNK> are <UNK> <END>
GT:<START> a salad bowl full of broccoli carrots <UNK> and <UNK> <UNK> <END>



12 INLINE QUESTION 2

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

Your Answer:

The parameters are Wx of shape (D, 4H), Wh of shape (H, 4H), and b of shape (4H,). So, the total number of parameters are D * 4H + H * 4H + 4H = 4H * (D+H+1).

13 INLINE QUESTION 3

What is the memory usage in the forward and backward pass of an Vanilla LSTM cell? Refer to lstm_step_forward and lstm_step_backward functions in ie590/rnn_layers.py

Your Answer:

In forward psss, we need to store the input x, parameters (Wx, Wh, b), and intermediate hidden variables (prev_h, prev_c). Depending on your coding methods, other intermediate variables like i, f, o, g may be freed after the cell is called, so, I would not count their storage here. As a result, the memory usage in forward pass is N * D + 4H * (D+H+1) + 2N * H. The backward pass requires the same storage because each above input x, parameters (Wx, Wh, b), and intermediate hidden variables (prev_h, prev_c) has a gradient counterpart of the same size as itself.