

这篇文章并不是介绍RNN的原理，而是在已经懂得原理的基础之上对其进行实现。











主要工作如下：

1. 图像特征提取；
2. 将单词转化为网络可以识别词向量；
3. 无时间序列的RNN（基础）；
4. 带有时间序列的RNN（升华）；
5. 激活函数（将RNN得到的隐藏状态转化为分数）；
6. 损失函数；
7. 将图像特征和词向量作为输入，进行网络训练。

1 数据说明

这里我们使用的是Microsoft coco数据集，这个数据集有80000张训练集和40000张验证集，其中每张图片有5个说明。

cs231n课程已经为我们提供接下来所需要的数据，我们看下都有什么内容：

 coco2014_captions.h5	2016/1/27 18:18	H5 文件	41,917 KB
 coco2014_vocab.json	2016/1/27 18:18	JSON 文件	24 KB
 train2014_images	2016/1/27 18:18	文本文档	4,366 KB
 train2014_urls	2016/1/27 18:18	文本文档	5,066 KB
 train2014_vgg16_fc7.h5	2016/1/26 17:04	H5 文件	1,324,535...
 train2014_vgg16_fc7_pca.h5	2016/2/7 16:58	H5 文件	165,569 KB
 val2014_images	2016/1/27 18:18	文本文档	1,978 KB
 val2014_urls	2016/1/27 18:18	文本文档	2,480 KB
 val2014_vgg16_fc7.h5	2016/1/26 17:29	H5 文件	648,071 KB
 val2014_vgg16_fc7_pca.h5	2016/2/7 16:58	H5 文件	81,011 KB

对于所有的图片，都是从VGG-16网络的第7层卷积层提取出来的特征，这些特征存储在 `train2014_vgg16_fc7.h5` 和 `val2014_vgg16_fc7.h5` 中。

为了节省处理的时间和内存要求，这些特征的维度得到了降低，从4096到512。这些特征存储在 `train_vgg16_fc7_pca.h5` 和 `val_vgg16_fc7_pca.h5` 中。

`train2014_urls.txt` 和 `val2014_urls.txt` 存储的是图片的链接，便于我们接下来的可视化图片和说明。

每个单词都被分配一个ID，这些ID被存储在 `coco2014_vocab.json` 文件中。

在词汇中我们增加了一些特殊的符号。`<START>` 和 `<END>` 分别表示说明的开始和结束；一些生僻词用 `<UNK>` 来代替。

因为我们想要训练不同长度的小批量数据，所以对于很短的说明我们在 `<END>` 后面增加了 `<NULL>`，但是对于 `<NULL>` 符号不计算它的损失函数和梯度。

一些前提已经介绍完了，下面我们来看下我们的数据。

我们从数据集中选1个来展示，如下：

`<START> a <UNK> old fire hydrant surrounded by <UNK> <END>`



2 Word embedding

使用计算机对自然语言进行处理，便需要将自然语言处理成为机器能够识别的符号，加上在机器学习过程中，需要将其进行数值化。而词是自然语言理解与处理的基础，因此需要对词进行数值化，词向量(Word Representation, Word embedding)便是一种可行又有效的方法。何为词向量，即使用一个指定长度的实数向量 v 来表示一个词。

词向量将作为我们的 x 进行学习和训练。

注意：Word embedding过程也涉及到参数的学习过程，因此也需要进行前向和反向传播。

关于Word embedding的详细介绍，可参阅[知乎提问](#)。

2.1 前向传播

代码如下：

```
1 def word_embedding_forward(x, W):
2     out, cache = None, None
3
4     #####
5     # TODO: Implement the forward pass for word embeddings.
6     #
7     #
8     N, T = x.shape
9     V, D = W.shape
10    out = np.zeros((N, T, D))
11
12    for i in range(N):
13        for j in range(T):
14            out[i, j] = W[x[i, j]]
15
16    cache = (x, W.shape)
17
18    #####
19    return out, cache
```

2.2 反向传播

前向传播是将单词转化为向量，而反向传播我们不能将向量转化为单词，这里我们仅仅返回Word embedding矩阵的梯度。

实现代码如下：

```
1 def word_embedding_backward(dout, cache):
2     dW = None
3
4     #####
5     # TODO: Implement the backward pass for word embeddings.
6     #
7     #
8     # HINT: Look up the function np.add.at
9     #
```

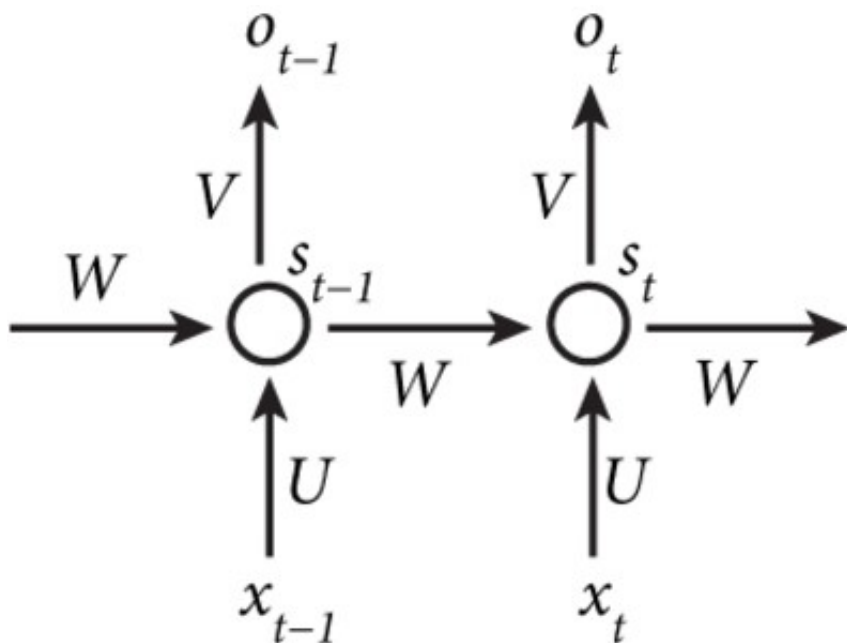
```

7
#####
#####
8     x, W_shape = cache
9     dW = np.zeros(W_shape)
10    np.add.at(dW, x, dout)
11    # pass
12
#####
#####
13    #                                END OF YOUR CODE
    #
14
#####
#####
15    return dW

```

3 单步运算

我们以一个神经元在 t 时刻的运算为例来分析。



3.1 前向传播

单步的前向传播大致过程如下：

s_t 为隐藏层的第 t 步的状态，它是网络的记忆单元。 s_t 根据当前输入层的输出与上一步隐藏层的状态进行计算。

$s_t = f(Ux_t + Ws_{t-1})$ ，其中 f 一般是非线性的激活函数，如tanh或ReLU，在计算 s_0 时，即第一个单词的隐藏层状态，需要用到 s_{-1} ，但是其并不存在，在实现中一般置为0向量；

这里我们使用的是tanh激活函数。

实现代码如下：

```
1 def rnn_step_forward(x, prev_h, Wx, Wh, b):
2
3     next_h, cache = None, None
4     #####
5     # TODO: Implement a single forward step for the vanilla RNN. Store the
    next #
6     # hidden state and any values you need for the backward pass in the
    next_h #
7     # and cache variables respectively.
    #
8     #####
9     a = prev_h.dot(Wh) + x.dot(Wx) + b
10    next_h = np.tanh(a)
11    cache = (x, prev_h, Wh, Wx, b, next_h)
12    #pass
13    #####
14    #                                END OF YOUR CODE
    #
15    #####
16    return next_h, cache
```

参数解释： `prev_h` (N,H)维数组，隐藏层第t-1步的状态，即上图的 S_{t-1} ；`x` (N,D)维数组，单步输入；`wx` (D,H)维数组，输入层对隐藏层的权重，即U；`wh` (H,H)维数组，隐藏层对隐藏层的权重，即W；`b` (H,)维数组。

3.2 反向传播

实现代码如下：

```
1 def rnn_step_backward(dnext_h, cache):
2
3     dx, dprev_h, dWx, dWh, db = None, None, None, None, None
4
5     #####
6     # TODO: Implement the backward pass for a single step of a vanilla
    RNN. #
```

```

6      #
      #
7      # HINT: For the tanh function, you can compute the local derivative
in terms #
8      # of the output value from tanh.
      #
9
#####
#####
10     x, prev_h, Wh, Wx, b, next_h = cache
11     da = dnext_h * (1 - next_h * next_h)
12     dx = da.dot(Wx.T)
13     dprev_h = da.dot(Wh.T)
14     dWx = x.T.dot(da)
15     dWh = prev_h.T.dot(da)
16     db = np.sum(da, axis=0)
17     # pass
18
#####
#####
19     #                                END OF YOUR CODE
      #
20
#####
#####
21     return dx, dprev_h, dWx, dWh, db

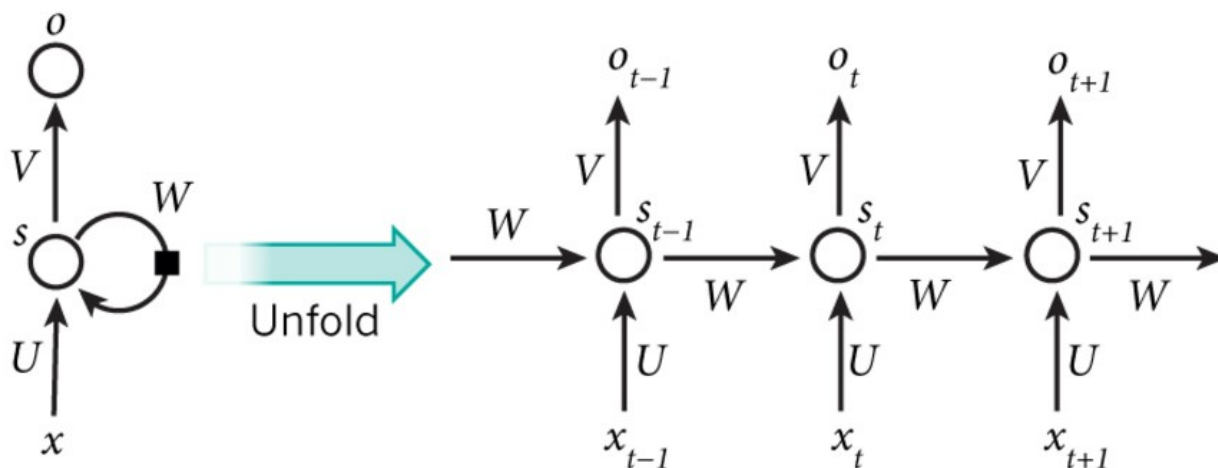
```

参数解释：反向传播的数组格式同前向传播。

这样我们就实现了RNN网络的一个单步过程，下面我们就可以使用单步来实现一个完整的RNN过程。

4 RNN

下面我们就在上节（单时刻）的基础之上实现一个完整的RNN（不同时刻）过程。



4.1 前向传播

不同时刻的前向传播就是单时刻的连续过程。

对于任意一个序列索引号 t ，我们隐藏状态 $h^{(t)}$ 由 $x^{(t)}$ 和 $h^{(t-1)}$ 得到：

$$h^{(t)} = \sigma(z^{(t)}) = \sigma(Ux^{(t)} + Wh^{(t-1)} + b)$$

实现代码如下：

```

1 def rnn_forward(x, h0, Wx, Wh, b):
2     h, cache = None, None
3
4     #####
5     # TODO: Implement forward pass for a vanilla RNN running on a
6     # sequence of #
7     # input data. You should use the rnn_step_forward function that you
8     # defined #
9     # above.
10    #
11    #####
12    #####
13    N, T, D = x.shape
14    (H,) = b.shape
15    h = np.zeros((N, T, H))
16    prev_h = h0
17    for t in range(T):
18        xt = x[:, t, :]
19        next_h, _ = rnn_step_forward(xt, prev_h, Wx, Wh, b)
20        prev_h = next_h

```

```

16         h[:, t, :] = prev_h
17         cache = (x, h0, Wh, Wx, b, h)
18         # pass
19
20         #####
21         #
22         #
23         #####
24         #####
25         return h, cache

```

解释：x (N,T,D)维数组，其中T表示不同第t步的输入；h0 初始状态；其余参数同单步运算的参数。

4.2 反向传播

实现代码如下：

```

1 def rnn_backward(dh, cache):
2     dx, dh0, dWx, dWh, db = None, None, None, None, None
3
4     #####
5     # TODO: Implement the backward pass for a vanilla RNN running an
6     # entire sequence of data. You should use the rnn_step_backward function
7     # that you defined above.
8     #
9     #####
10
11     x, h0, Wh, Wx, b, h = cache
12     N, T, H = dh.shape
13     _, _, D = x.shape
14
15     next_h = h[:, T - 1, :]
16
17     dprev_h = np.zeros((N, H))
18     dx = np.zeros((N, T, D))
19     dh0 = np.zeros((N, H))
20     dWx = np.zeros((D, H))

```



```

18     dWx = np.zeros((H, H))
19     db = np.zeros((H,))
20
21     for t in range(T):
22         t = T - 1 - t
23         xt = x[:, t, :]
24
25         if t == 0:
26             prev_h = h0
27         else:
28             prev_h = h[:, t - 1, :]
29
30         step_cache = (xt, prev_h, Wh, Wx, b, next_h)
31         next_h = prev_h
32         dnext_h = dh[:, t, :] + dprev_h
33         dx[:, t, :], dprev_h, dWxt, dWht, dbt =
rnn_step_backward(dnext_h, step_cache)
34         dWx, dWx, db = dWx + dWxt, dWx + dWht, db + dbt
35
36     dh0 = dprev_h
37     # pass
38
39     #####
40     #
41     #
42     #####
43     return dx, dh0, dWx, dWx, db

```

整个的反向传播过程类似于全连接神经网络，这里我们就不对代码进行解释。

这样我们就实现了RNN不同时刻输入到隐藏层的状态。

注意：我们这里都只是计算到隐藏层在第 t 步的状态（ a_t ），还没有计算第 t 步的输出，即 O_t 。

5 激活函数层

还记得在RNN中我们说过，我们只是实现了输入层状态到隐藏层状态的一个实现，而要得到最后的输出（一般为分数）还需要经过激活函数的计算。

这部分在前面两次作业中已经较为详细的叙述，这里我们就较为省略了。

5.1 前向传播

$$o^{(t)} = Vh^{(t)} + c$$

实现代码如下：

```
1 def temporal_affine_forward(x, w, b):
2     N, T, D = x.shape
3     M = b.shape[0]
4     out = x.reshape(N * T, D).dot(w).reshape(N, T, M) + b
5     cache = x, w, b, out
6     return out, cache
7
```

5.2 反向传播

实现代码如下：

```
1 def temporal_affine_backward(dout, cache):
2     x, w, b, out = cache
3     N, T, D = x.shape
4     M = b.shape[0]
5
6     dx = dout.reshape(N * T, M).dot(w.T).reshape(N, T, D)
7     dw = dout.reshape(N * T, M).T.dot(x.reshape(N * T, D)).T
8     db = dout.sum(axis=(0, 1))
9
10    return dx, dw, db
11
```

6 softmax 损失函数

这里的损失函数和前面作业的softmax是类似的，但还是有些不同。

还记得前面我们对数据的说明吗？为了让每一个说明都有相同的长度，我们在每个说明后面引入了 `<NULL>` 字符，但是这些字符不应该参与损失函数的计算。

因此，我们的预测值和标签和需要接收一个 `mask` 数组来判断对哪个元素计算损失。

实现代码如下：

```

1 def temporal_softmax_loss(x, y, mask, verbose=False):
2     N, T, V = x.shape
3
4     x_flat = x.reshape(N * T, V)
5     y_flat = y.reshape(N * T)
6     mask_flat = mask.reshape(N * T)
7
8     probs = np.exp(x_flat - np.max(x_flat, axis=1, keepdims=True))
9     probs /= np.sum(probs, axis=1, keepdims=True)
10    loss = -np.sum(mask_flat * np.log(probs[np.arange(N * T), y_flat])) /
11    N
12    dx_flat = probs.copy()
13    dx_flat[np.arange(N * T), y_flat] -= 1
14    dx_flat /= N
15    dx_flat *= mask_flat[:, None]
16
17    if verbose:
18        print('dx_flat: ', dx_flat.shape)
19
20    dx = dx_flat.reshape(N, T, V)
21
22    return loss, dx

```

参数说明： `x` (N,T,V)维数组，经过激活函数层得到的分数； `y` (N,T)维数组，标签，其中 $0 \leq y[i, t] < V$ ； `mask` (N,T)维数组，其中 `mask[i,t]` 来判断 `x` 是否应该被用于计算损失。

就这样，我们已经完成了RNN网络所需要的全部内容。

7 模型构建

下面我们就使用上面的基础层来完成对RNN网络的构建。

代码如下：

```

1 class CaptioningRNN(object):
2     def __init__(self, word_to_idx, input_dim=512, wordvec_dim=128,
3                 hidden_dim=128, cell_type='rnn', dtype=np.float32): #1
4         if cell_type not in {'rnn', 'lstm'}:
5             raise ValueError('Invalid cell_type "%s"' % cell_type)
6
7         self.cell_type = cell_type
8         self.dtype = dtype
9         self.word_to_idx = word_to_idx

```

```

10         self.idx_to_word = {i: w for w, i in
word_to_idx.iteritems()}
11         self.params = {}
12
13         vocab_size = len(word_to_idx)
14
15         self._null = word_to_idx['<NULL>']
16         self._start = word_to_idx.get('<START>', None)
17         self._end = word_to_idx.get('<END>', None)
18
19         # Initialize word vectors
20         self.params['W_embed'] = np.random.randn(vocab_size,
wordvec_dim)
21         self.params['W_embed'] /= 100
22
23         # Initialize CNN -> hidden state projection parameters
24         self.params['W_proj'] = np.random.randn(input_dim,
hidden_dim)
25         self.params['W_proj'] /= np.sqrt(input_dim)
26         self.params['b_proj'] = np.zeros(hidden_dim)
27
28         # Initialize parameters for the RNN
29         dim_mul = {'lstm': 4, 'rnn': 1}[cell_type]
30         self.params['Wx'] = np.random.randn(wordvec_dim, dim_mul *
hidden_dim)
31         self.params['Wx'] /= np.sqrt(wordvec_dim)
32         self.params['Wh'] = np.random.randn(hidden_dim, dim_mul *
hidden_dim)
33         self.params['Wh'] /= np.sqrt(hidden_dim)
34         self.params['b'] = np.zeros(dim_mul * hidden_dim)
35
36         # Initialize output to vocab weights
37         self.params['W_vocab'] = np.random.randn(hidden_dim,
vocab_size)
38         self.params['W_vocab'] /= np.sqrt(hidden_dim)
39         self.params['b_vocab'] = np.zeros(vocab_size)
40
41         # Cast parameters to correct dtype
42         for k, v in self.params.iteritems():
43             self.params[k] = v.astype(self.dtype)
44
45         def loss(self, features, captions):#2
46             # Cut captions into two pieces: captions_in has
everything but the last word
47             # and will be input to the RNN; captions_out has
everything but the first

```

```

48         # word and this is what we will expect the RNN to
generate. These are offset
49         # by one relative to each other because the RNN should
produce word (t+1)
50         # after receiving word t. The first element of
captions_in will be the START
51         # token, and the first element of captions_out will be
the first word.
52         captions_in = captions[:, :-1]
53         captions_out = captions[:, 1:]
54
55         # You'll need this
56         mask = (captions_out != self._null)
57
58         # Weight and bias for the affine transform from image
features to initial
59         # hidden state
60         W_proj, b_proj = self.params['W_proj'],
self.params['b_proj']
61
62         # Word embedding matrix
63         W_embed = self.params['W_embed']
64
65         # Input-to-hidden, hidden-to-hidden, and biases for the
RNN
66         Wx, Wh, b = self.params['Wx'], self.params['Wh'],
self.params['b']
67
68         # Weight and bias for the hidden-to-vocab
transformation.
69         W_vocab, b_vocab = self.params['W_vocab'],
self.params['b_vocab']
70
71         loss, grads = 0.0, {}
72
73         #####
74         # TODO: Implement the forward and backward passes for
the CaptioningRNN. #
75         # 2.1
affine_out, affine_cache = affine_forward(features,
W_proj, b_proj)
76         # 2.2
word_embedding_out, word_embedding_cache =
word_embedding_forward(captions_in, W_embed)
77
78         # 2.3

```

```

79         if self.cell_type == 'rnn':
80             rnn_or_lstm_out, rnn_cache =
rnn_forward(word_embedding_out, affine_out, Wx, Wh, b)
81         elif self.cell_type == 'lstm':
82             rnn_or_lstm_out, lstm_cache =
lstm_forward(word_embedding_out, affine_out, Wx, Wh, b)
83         else:
84             raise ValueError('Invalid cell_type "%s"' %
self.cell_type)
85         # 2.4
86         temporal_affine_out, temporal_affine_cache =
temporal_affine_forward(rnn_or_lstm_out, W_vocab, b_vocab)
87         # 2.5
88         loss, dtemporal_affine_out =
temporal_softmax_loss(temporal_affine_out, captions_out, mask)
89         # 2.6
90         drnn_or_lstm_out, grads['W_vocab'], grads['b_vocab'] =
temporal_affine_backward(dtemporal_affine_out,
91
temporal_affine_cache)
92         # 2.7
93         if self.cell_type == 'rnn':
94             dword_embedding_out, daffine_out, grads['Wx'],
grads['Wh'], grads['b'] = rnn_backward(
95                 drnn_or_lstm_out, rnn_cache)
96             elif self.cell_type == 'lstm':
97                 dword_embedding_out, daffine_out, grads['Wx'],
grads['Wh'], grads['b'] = lstm_backward(
98                     drnn_or_lstm_out, lstm_cache)
99         else:
100             raise ValueError('Invalid cell_type "%s"' %
self.cell_type)
101         # 2.8
102         grads['W_embed'] =
word_embedding_backward(dword_embedding_out, word_embedding_cache)
103         # 2.9
104         dfeatures, grads['W_proj'], grads['b_proj'] =
affine_backward(daffine_out, affine_cache)
105         # pass
106
#####
#####
107         #
#

```

```

108 #####
109 #####
110         return loss, grads
111
112     def sample(self, features, max_length=30): #3
113         N = features.shape[0]
114         captions = self._null * np.ones((N, max_length),
dtype=np.int32)
115
116         # Unpack parameters
117         W_proj, b_proj = self.params['W_proj'],
self.params['b_proj']
118         W_embed = self.params['W_embed']
119         Wx, Wh, b = self.params['Wx'], self.params['Wh'],
self.params['b']
120         W_vocab, b_vocab = self.params['W_vocab'],
self.params['b_vocab']
121
122 #####
123 #####
124         # TODO: Implement test-time sampling for the model. You
will need to #
125         N, D = features.shape
126         affine_out, affine_cache = affine_forward(features,
W_proj, b_proj)
127
128         prev_word_idx = [self._start] * N
129         prev_h = affine_out
130         prev_c = np.zeros(prev_h.shape)
131         captions[:, 0] = self._start
132         for i in range(1, max_length):
133             prev_word_embed = W_embed[prev_word_idx]
134             if self.cell_type == 'rnn':
135                 next_h, rnn_step_cache =
rnn_step_forward(prev_word_embed, prev_h, Wx, Wh, b)
136             elif self.cell_type == 'lstm':
137                 next_h, next_c, lstm_step_cache =
lstm_step_forward(prev_word_embed, prev_h, prev_c, Wx, Wh, b)
138             prev_c = next_c
139             else:
140                 raise ValueError('Invalid cell_type "%s"' %
self.cell_type)

```

```

140         vocab_affine_out, vocab_affine_out_cache =
affine_forward(next_h, W_vocab, b_vocab)
141         captions[:, i] = list(np.argmax(vocab_affine_out,
axis=1))
142         prev_word_idx = captions[:, i]
143         prev_h = next_h
144         # pass
145
#####
#####
146         #                               END OF YOUR CODE
#
147
#####
#####
148         return captions

```

代码解释：

1# 参数初始化。 `word_to_idx` 字典，存储词以及它所对应的值； `input_dim` 输入图片的特征向量； `wordvec_dim` 词向量； `hidden_dim` RNN网络的隐含层； `cell_type` 确定网络类型，即RNN或者LSTM（这个我们在下一节会实现）。

2# 输入图像特征向量和标签（说明），经过RNN计算损失函数和参数的梯度，即训练。

这里 `captions_in` 被送进RNN网络，它拥有除了最后一个单词的所有， `captions_out` 是我们期望获得的，它拥有除了第一个单词的所有。

2.1# 将图像特征输入映射层来初始化；2.2# 将单词转化为词向量；2.3# 将初始化的图像和词向量输入RNN，得到各个时刻隐含层的输出。

2.4# 经过激活函数层得到分数。

2.5# ——2.9# 反向传播。

3# 预测过程。

预测过程即是为特征向量挑选最好的说明的过程。

这样我们就完成了RNN模型的搭建工作。

8 实验分析

万事俱备，只欠东风。下面我们就可以使用前文的数据和模型来进行“看图说话”了。

8.1 训练

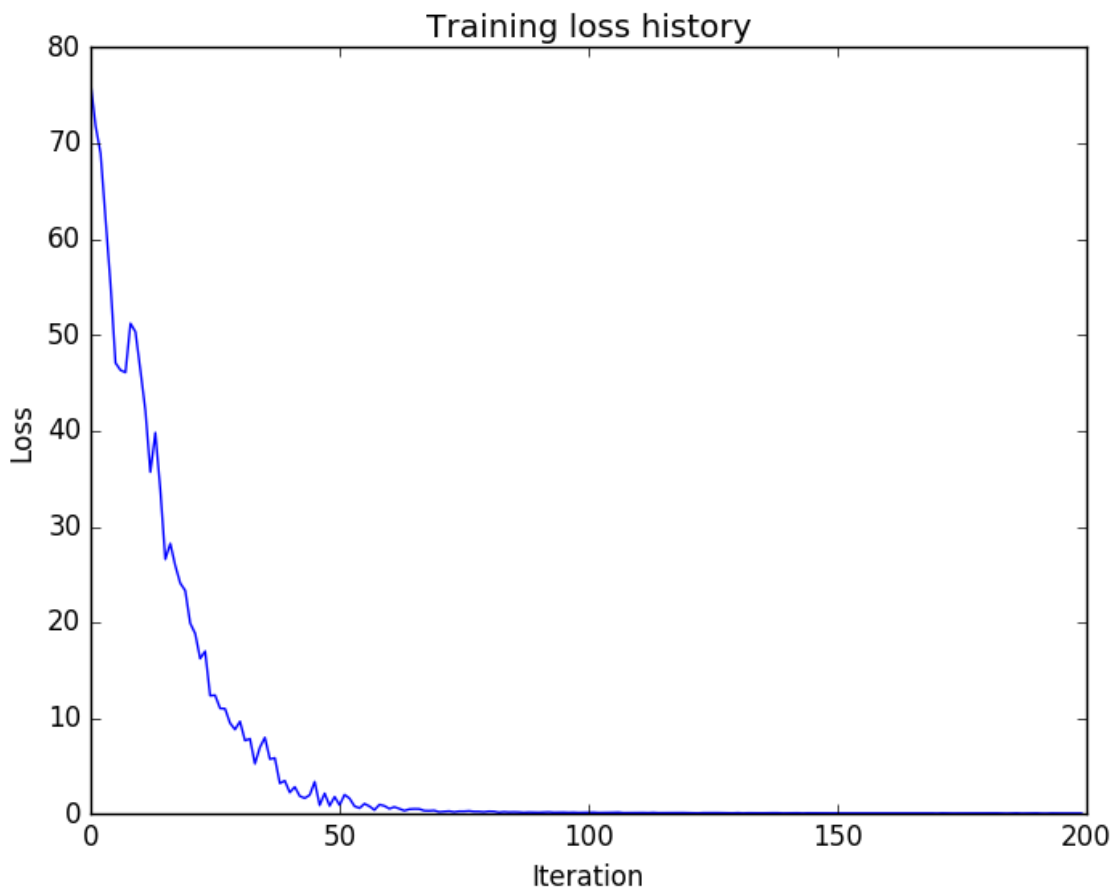
为了加快速度，我们只使用100个数据来训练。

注意： `CaptioningSolver` 这里没有进行展开，它和前面作业的 `Solver` 是同样的效果，这里输入数据、模型、优化器等参数，即可进行训练。

代码如下：

```
1 small_data = load_coco_data(max_train=100)
2
3 small_rnn_model = CaptioningRNN(
4     cell_type='rnn',
5     word_to_idx=data['word_to_idx'],
6     input_dim=data['train_features'].shape[1],
7     hidden_dim=512,
8     wordvec_dim=256,
9 )
10
11 small_rnn_solver = CaptioningSolver(small_rnn_model, small_data,
12     update_rule='adam',
13     num_epochs=50,
14     batch_size=25,
15     optim_config={
16         'learning_rate': 5e-3,
17     },
18     lr_decay=0.95,
19     verbose=True, print_every=10,
20 )
21
22 small_rnn_solver.train()
23
24 # Plot the training losses
25 plt.plot(small_rnn_solver.loss_history)
26 plt.xlabel('Iteration')
27 plt.ylabel('Loss')
28 plt.title('Training loss history')
29 plt.show()
```

我们得到的loss函数图如下：



loss值还是还是很可观的。

接下来我们就使用保存好的参数进行预测了。

8.2 预测

代码如下：

```
1 for split in ['train', 'val']:
2     minibatch = sample_coco_minibatch(small_data, split=split,
3     batch_size=2)
4     gt_captions, features, urls = minibatch
5     gt_captions = decode_captions(gt_captions, data['idx_to_word'])
6
7     sample_captions = small_rnn_model.sample(features)
8     sample_captions = decode_captions(sample_captions, data['idx_to_word'])
9
10    for gt_caption, sample_caption, url in zip(gt_captions,
11    sample_captions, urls):
12        plt.imshow(image_from_url(url))
13        plt.title('%s\n%s\nGT:%s' % (split, sample_caption, gt_caption))
14        plt.axis('off')
15        plt.show()
```

我们看看其中一个结果：

val
<START> three little girl riding horses across the beach where people are swimming <END>
GT:<START> surfers are sitting and standing on their surfboards in the water <END>



9 总结

这次作业主要实现了RNN基础网络的构建，并通过VGG提取的图像特征做出了图像说明。

与传统的神经网络相比，RNN主要有以下两点不同：

1. 隐藏层的状态不仅与上一层的状态有关，还取决于该层上一时刻的状态；
2. RNN每一层的参数是共享的。