Neural_Machine_Translation_Pytorch

June 14, 2019

In [1]:
```
import unicodedata
import string
import re
import random
import time
import math

import torch
import torch.nn as nn
from torch.autograd import Variable
from torch import optim
import torch.nn.functional as F
```

In [3]:
```
USE_CUDA = True

SOS_token = 0
EOS_token = 1

class Lang:
    def __init__(self, name):
        self.name = name
        self.word2index = {}
        self.word2count = {}
        self.index2word = {0: "SOS", 1: "EOS"}
        self.n_words = 2 # Count SOS and EOS

    def index_words(self, sentence):
        for word in sentence.split(' '):
            self.index_word(word)

    def index_word(self, word):
        if word not in self.word2index:
            self.word2index[word] = self.n_words
            self.word2count[word] = 1
            self.index2word[self.n_words] = word
            self.n_words += 1
        else:
            self.word2count[word] += 1
```

```
In [4]: """
        PRE-PROCESS THE TEXT:
        """

        # Turn a Unicode string to plain ASCII, thanks to http://stackoverflow.com/a/518232/2809
        def unicode_to_ascii(s):
            return ''.join(
                c for c in unicodedata.normalize('NFD', s)
                if unicodedata.category(c) != 'Mn'
            )

        # Lowercase, trim, and remove non-letter characters
        def normalize_string(s):
            s = unicode_to_ascii(s.lower().strip())
            s = re.sub(r"([.!?])", r" \1", s)
            s = re.sub(r"[^a-zA-Z.!?]+", r" ", s)
            return s

In [5]: def read_langs(lang1, lang2, reverse=False):
            print("Reading lines...")

            # Read the file and split into lines
            lines = open('data/%s-%s.txt' % (lang1, lang2)).read().strip().split('\n')

            # Split every line into pairs and normalize
            pairs = [[normalize_string(s) for s in l.split('\t')] for l in lines]

            # Reverse pairs, make Lang instances
            if reverse:
                pairs = [list(reversed(p)) for p in pairs]
                input_lang = Lang(lang2)
                output_lang = Lang(lang1)
            else:
                input_lang = Lang(lang1)
                output_lang = Lang(lang2)

            return input_lang, output_lang, pairs

In [6]: MAX_LENGTH = 10

        good_prefixes = (
            "i am ", "i m ",
            "he is", "he s ",
            "she is", "she s",
            "you are", "you re "
        )

        def filter_pair(p):
```

```
            return len(p[0].split(' ')) < MAX_LENGTH and len(p[1].split(' ')) < MAX_LENGTH and \
                p[1].startswith(good_prefixes)

        def filter_pairs(pairs):
            return [pair for pair in pairs if filter_pair(pair)]

        def prepare_data (lang1_name, lang2_name, reverse=False):
            input_lang, output_lang, pairs = read_langs(lang1_name, lang2_name, reverse)
            print("Read %s sentence pairs" % len(pairs))

            pairs = filter_pairs(pairs)
            print("Trimmed to %s sentence pairs" % len(pairs))

            print("Indexing words...")
            for pair in pairs:
                input_lang.index_words(pair[0])
                output_lang.index_words(pair[1])

            return input_lang, output_lang, pairs

        input_lang, output_lang, pairs = prepare_data('eng', 'fra', True)

        # Print an example pair
        print(random.choice(pairs))
```

```
Reading lines...
Read 167130 sentence pairs
Trimmed to 10724 sentence pairs
Indexing words...
['tu n es pas le seul canadien ici .', 'you re not the only canadian here .']
```

```
In [9]: pair = random.choice(pairs)
        print(pair[0], '\t', pair[1])
        print(variable_from_sentence(input_lang, pair[0]))
        print(variable_from_sentence(output_lang, pair[1]))
```

```
il cherche actuellement un nouveau poste .          he is seeking a new position .
tensor([[  25],
        [2170],
        [ 410],
        [  82],
        [ 479],
        [1286],
        [   5],
        [   1]], device='cuda:0')
tensor([[  15],
        [  47],
```

```
        [1807],
        [  49],
        [ 295],
        [1319],
        [   4],
        [   1]], device='cuda:0')
```

```
In [10]: # Return a list of indexes, one for each word in the sentence
         def indexes_from_sentence(lang, sentence):
             return [lang.word2index[word] for word in sentence.split(' ')]

         # this function returns a list of word indexes as a torch tensor:
         def variable_from_sentence(lang, sentence):
             indexes = indexes_from_sentence(lang, sentence)
             indexes.append(EOS_token)
             var = Variable(torch.LongTensor(indexes).view(-1, 1))
         #     print('var =', var)
             if USE_CUDA: var = var.cuda()
             return var

         def variables_from_pair(pair):
             input_variable = variable_from_sentence(input_lang, pair[0])
             target_variable = variable_from_sentence(output_lang, pair[1])
             return (input_variable, target_variable)
```

```
In [11]: indexes_from_sentence(output_lang, 'i am going to get some good sleep')
```

```
Out[11]: [2, 17, 72, 550, 1706, 2185, 51, 1038]
```

```
In [12]: pairs[1001]
```

```
Out[12]: ['je ne suis pas naif .', 'i m not naive .']
```

```
In [13]: # transcode words into a LIST of integers corresponding to word codes [according to the
         indexes_from_sentence(input_lang, 'je n aime pas lire')
```

```
Out[13]: [6, 259, 1479, 260, 608]
```

```
In [14]: # transcode words into a TORCH TENSOR of integers corresponding to word codes [accordin
         variable_from_sentence(output_lang, 'i am good')
```

```
Out[14]: tensor([[ 2],
                 [17],
                 [51],
                 [ 1]], device='cuda:0')
```

```
In [15]: # get a 2-tuple of torch tensors from a source-target pair:
         variables_from_pair(pairs[1000])
```

```
Out[15]: (tensor([[  6],
                 [316],
                 [ 11],
                 [260],
                 [ 15],
                 [126],
                 [ 53],
                 [127],
                 [  5],
                 [  1]], device='cuda:0'), tensor([[  2],
                 [  3],
                 [157],
                 [ 75],
                 [  4],
                 [  1]], device='cuda:0'))
```

```
In [16]: print(output_lang.name)
         print(output_lang.index2word[100])
         print(output_lang.word2index['buying'])
```

```
eng
buying
100
```

```
In [17]: class EncoderRNN(nn.Module):
             def __init__(self, input_size, hidden_size, n_layers=1):
                 super(EncoderRNN, self).__init__()

                 self.input_size = input_size
                 self.hidden_size = hidden_size
                 self.n_layers = n_layers

                 self.embedding = nn.Embedding(input_size, hidden_size)
                 self.gru = nn.GRU(hidden_size, hidden_size, n_layers)

             def forward(self, word_inputs, hidden):
                 # Note: we run this all at once (over the whole input sequence)
                 seq_len = len(word_inputs)
                 embedded = self.embedding(word_inputs).view(seq_len, 1, -1)
                 output, hidden = self.gru(embedded, hidden)
                 return output, hidden

             def init_hidden(self):
                 hidden = Variable(torch.zeros(self.n_layers, 1, self.hidden_size))
                 if USE_CUDA: hidden = hidden.cuda()
                 return hidden
```

```
In [18]: input_size = 10
         hidden_size = 10
```

```
n_layers = 1
a = EncoderRNN(input_size, hidden_size, n_layers=n_layers)
a
```

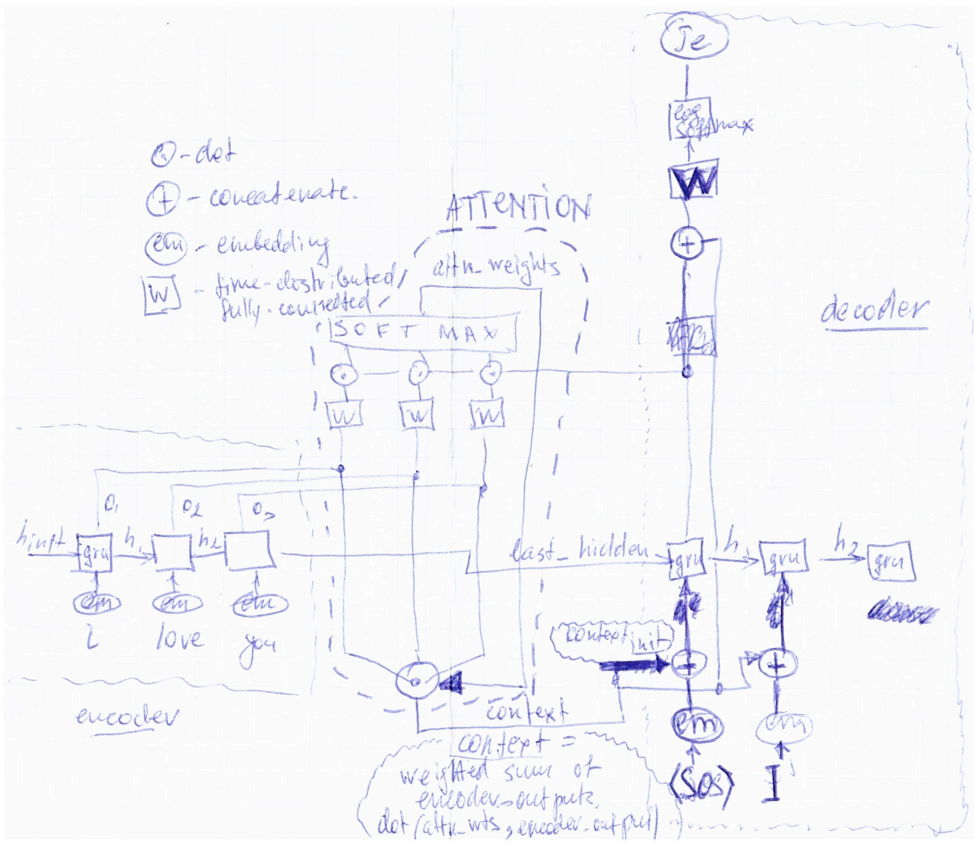
Out[18]:
```
EncoderRNN(
  (embedding): Embedding(10, 10)
  (gru): GRU(10, 10)
)
```

In [19]:
```
class BahdanauAttnDecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size, n_layers=1, dropout_p=0.1):
        super(AttnDecoderRNN, self).__init__()

        # Define parameters
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.n_layers = n_layers
        self.dropout_p = dropout_p
        self.max_length = max_length

        # Define layers
        self.embedding = nn.Embedding(output_size, hidden_size)
        self.dropout = nn.Dropout(dropout_p)
        self.attn = GeneralAttn(hidden_size)
        self.gru = nn.GRU(hidden_size * 2, hidden_size, n_layers, dropout=dropout_p)
        self.out = nn.Linear(hidden_size, output_size)

    def forward(self, word_input, last_hidden, encoder_outputs):
        # Note that we will only be running forward for a single decoder time step, but

        # Get the embedding of the current input word (last output word)
        word_embedded = self.embedding(word_input).view(1, 1, -1) # S=1 x B x N
        word_embedded = self.dropout(word_embedded)

        # Calculate attention weights and apply to encoder outputs
        attn_weights = self.attn(last_hidden[-1], encoder_outputs)
        context = attn_weights.bmm(encoder_outputs.transpose(0, 1)) # B x 1 x N

        # Combine embedded input word and attended context, run through RNN
        rnn_input = torch.cat((word_embedded, context), 2)
        output, hidden = self.gru(rnn_input, last_hidden)

        # Final output layer
        output = output.squeeze(0) # B x N
        output = F.log_softmax(self.out(torch.cat((output, context), 1)))

        # Return final output, hidden state, and attention weights (for visualization)
        return output, hidden, attn_weights
```

```
In [20]: class Attn(nn.Module):
             def __init__(self, method, hidden_size, max_length=MAX_LENGTH):
                 super(Attn, self).__init__()

                 self.method = method
                 self.hidden_size = hidden_size

                 if self.method == 'general':
                     self.attn = nn.Linear(self.hidden_size, hidden_size)

                 elif self.method == 'concat':
                     self.attn = nn.Linear(self.hidden_size * 2, hidden_size)
                     self.other = nn.Parameter(torch.FloatTensor(1, hidden_size))

             def forward(self, hidden, encoder_outputs):
                 seq_len = len(encoder_outputs)

                 # Create variable to store attention energies
                 attn_energies = Variable(torch.zeros(seq_len)) # B x 1 x S
                 if USE_CUDA: attn_energies = attn_energies.cuda()

                 # Calculate energies for each encoder output
                 for i in range(seq_len):
                     attn_energies[i] = self.score(hidden, encoder_outputs[i])

                 # Normalize energies to weights in range 0 to 1, resize to 1 x 1 x seq_len
                 return F.softmax(attn_energies, dim=0).unsqueeze(0).unsqueeze(0)

             def score(self, hidden, encoder_output):

                 if self.method == 'dot':
                     energy = hidden.dot(encoder_output)
                     return energy

                 elif self.method == 'general':
                     energy = self.attn(encoder_output) # to get attention energies, we FORWARD
                     energy = hidden.flatten().dot(energy.flatten()) # and project the result on
                     return energy

                 elif self.method == 'concat':
                     energy = self.attn(torch.cat((hidden, encoder_output), 1))
                     energy = self.other.dot(energy)
                     return energy
```

0.1 seq2seq with attention

```
In [21]: class AttnDecoderRNN(nn.Module):
             """
```



- dot
- concatenate.
- embedding
- time-distributed/fully-connected
ATTENTION
attn_weights
SOFTMAX
encoder
decoder
last_hidden
context
context = weighted sum of encoder_outputs, dot(attn_wts, encoder_output)
I
love
you
Je
I

title

```python
    REMEMBER, IN PYTORCH RNN CELLS (BY DEFAULT) INPUT TENSORS HAVE SHAPE (seq_len, batc
    """
    def __init__(self, attn_model, hidden_size, output_size, n_layers=1, dropout_p=0.1)
        super(AttnDecoderRNN, self).__init__()

        # Keep parameters for reference
        self.attn_model = attn_model         # either 'general', or 'concat'
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.n_layers = n_layers
        self.dropout_p = dropout_p

        # Define layers
        self.embedding = nn.Embedding(output_size, hidden_size)
        self.gru = nn.GRU(hidden_size * 2, hidden_size, n_layers, dropout=dropout_p)
        self.out = nn.Linear(hidden_size * 2, output_size)

        # Choose attention model
        if attn_model != 'none':
            self.attn = Attn(attn_model, hidden_size)

    def forward(self, word_input, last_context, last_hidden, encoder_outputs):
        """
        Note: we run this one step (one word) at a time
        we forward one word at a time. last_context is updated every forward pass.
        on the first pass, last_hidden is the last hidden state of the ENCODER, then hi
        on every forward pass.
        encoder_outputs are not updated by the decoder (obviously).
        """

        # Get the embedding of the current input word (last output word)
        word_embedded = self.embedding(word_input).view(1, 1, -1) # S = 1 x Batch_size

        # Combine embedded input word and last context, run through RNN
        rnn_input = torch.cat((word_embedded, last_context.unsqueeze(0)), 2) # torch.Si
        rnn_output, hidden = self.gru(rnn_input, last_hidden)                # torch.Si

        # Calculate attention from current RNN state and all encoder outputs; apply to
        attn_weights = self.attn(rnn_output.squeeze(0), encoder_outputs)     # torch.Si

        """ ##########################################################
        !!! CONTEXT - is a weighted sum of all the _encoder_ outputs !!!
        """
        context = attn_weights.bmm(encoder_outputs.transpose(0, 1)) # B x 1 x N      #

        # Final output layer (next word prediction) using the RNN hidden state and cont
        rnn_output = rnn_output.squeeze(0) # S=1 x B x N -> B x N
        context = context.squeeze(1)       # B x S=1 x N -> B x N
```

```
            '''why log-softmax???????????????????????????????????'''
            output = F.log_softmax(self.out(torch.cat((rnn_output, context), 1)), dim=1) #

            # Return final output, hidden state, and attention weights (for visualization)
            return output, context, hidden, attn_weights
```

In [22]:
```
n_layers = 2
encoder_test = EncoderRNN(10, 10, n_layers=n_layers)
decoder_test = AttnDecoderRNN('general', 10, 10, n_layers=n_layers)
print(encoder_test)
print(decoder_test)

"""
First, we forward our sequence through the encoder to get its outputs and the last hidd
"""
encoder_hidden = encoder_test.init_hidden()         # we need to feed something as a hi
word_input = Variable(torch.LongTensor([1, 2, 3]))
if USE_CUDA:
    encoder_test.cuda()
    word_input = word_input.cuda()
encoder_outputs, encoder_hidden = encoder_test(word_input, encoder_hidden)

"""
Second, we forward through the decoder the (1) same word embedding sequence, (2) decode
plus the (3) decoder_hidden state (for the first word in the sequence the decoder_hidde
the encoder_hidden, i.e. last hidden state of the encoder) and (4) last_context (for th
the context is initialized to zeros, then updated at every time step/word).
"""
word_inputs = Variable(torch.LongTensor([1, 2, 3]))
decoder_attns = torch.zeros(1, 3, 3)
decoder_hidden = encoder_hidden

# for the first run, we need to initialize the decoder context. The decoder_context is
# time we forward a word through the decoder (see the loop below):
decoder_context = Variable(torch.zeros(1, decoder_test.hidden_size))

if USE_CUDA:
    decoder_test.cuda()
    word_inputs = word_inputs.cuda()
    decoder_context = decoder_context.cuda()


for i in range(3):
    decoder_output, decoder_context, decoder_hidden, decoder_attn = decoder_test(word_i
    print(decoder_output.size(), decoder_hidden.size(), decoder_attn.size())
    decoder_attns[0, i] = decoder_attn.squeeze(0).cpu().data
```

```
EncoderRNN(
```

```
  (embedding): Embedding(10, 10)
  (gru): GRU(10, 10, num_layers=2)
)
AttnDecoderRNN(
  (embedding): Embedding(10, 10)
  (gru): GRU(20, 10, num_layers=2, dropout=0.1)
  (out): Linear(in_features=20, out_features=10, bias=True)
  (attn): Attn(
    (attn): Linear(in_features=10, out_features=10, bias=True)
  )
)
torch.Size([1, 10]) torch.Size([2, 1, 10]) torch.Size([1, 1, 3])
torch.Size([1, 10]) torch.Size([2, 1, 10]) torch.Size([1, 1, 3])
torch.Size([1, 10]) torch.Size([2, 1, 10]) torch.Size([1, 1, 3])
```

In [23]:
```
teacher_forcing_ratio = 0.5
clip = 5.0

def train(input_variable, target_variable, encoder, decoder, encoder_optimizer, decoder

    # Zero gradients of both optimizers
    encoder_optimizer.zero_grad()
    decoder_optimizer.zero_grad()
    loss = 0 # Added onto for each word

    # Get size of input and target sentences
    input_length = input_variable.size()[0]
    target_length = target_variable.size()[0]

    # Run words through encoder
    encoder_hidden = encoder.init_hidden()
    encoder_outputs, encoder_hidden = encoder(input_variable, encoder_hidden)

    # Prepare input and output variables
    decoder_input = Variable(torch.LongTensor([[SOS_token]]))
    decoder_context = Variable(torch.zeros(1, decoder.hidden_size))
    decoder_hidden = encoder_hidden # Use last hidden state from encoder to start decod
    if USE_CUDA:
        decoder_input = decoder_input.cuda()
        decoder_context = decoder_context.cuda()

    # Choose whether to use teacher forcing
    use_teacher_forcing = random.random() < teacher_forcing_ratio
    if use_teacher_forcing:

        # Teacher forcing: Use the ground-truth target as the next input
        for di in range(target_length):
```

```
                decoder_output, decoder_context, decoder_hidden, decoder_attention = decode
                loss += criterion(decoder_output, target_variable[di])
                decoder_input = target_variable[di] # Next target is next input

        else:
            # Without teacher forcing: use network's own prediction as the next input
            for di in range(target_length):
                decoder_output, decoder_context, decoder_hidden, decoder_attention = decode
                loss += criterion(decoder_output, target_variable[di])

                # Get most likely word index (highest value) from output
                topv, topi = decoder_output.data.topk(1)
                ni = topi[0][0]

                decoder_input = Variable(torch.LongTensor([[ni]])) # Chosen word is next in
                if USE_CUDA: decoder_input = decoder_input.cuda()

                # Stop at end of sentence (not necessary when using known targets)
                if ni == EOS_token: break

        # Backpropagation
        loss.backward()
        torch.nn.utils.clip_grad_norm_(encoder.parameters(), clip)
        torch.nn.utils.clip_grad_norm_(decoder.parameters(), clip)
        encoder_optimizer.step()
        decoder_optimizer.step()

    #     return loss.data[0] / target_length
        return loss.item() / target_length
```

In [24]:
```
def as_minutes(s):
    m = math.floor(s / 60)
    s -= m * 60
    return '%dm %ds' % (m, s)

def time_since(since, percent):
    now = time.time()
    s = now - since
    es = s / (percent)
    rs = es - s
    return '%s (- %s)' % (as_minutes(s), as_minutes(rs))
```

In [25]:
```
attn_model = 'general'
hidden_size = 500
n_layers = 2
dropout_p = 0.05

# Initialize models
```

```
encoder = EncoderRNN(input_lang.n_words, hidden_size, n_layers)
decoder = AttnDecoderRNN(attn_model, hidden_size, output_lang.n_words, n_layers, dropou

# Move models to GPU
if USE_CUDA:
    encoder.cuda()
    decoder.cuda()

# Initialize optimizers and criterion
learning_rate = 0.0001
encoder_optimizer = optim.Adam(encoder.parameters(), lr=learning_rate)
decoder_optimizer = optim.Adam(decoder.parameters(), lr=learning_rate)
criterion = nn.NLLLoss()
```

In [26]:
```
# Configuring training
n_epochs = 50000
plot_every = 200
print_every = 1000

# Keep track of time elapsed and running averages
start = time.time()
plot_losses = []
print_loss_total = 0 # Reset every print_every
plot_loss_total = 0 # Reset every plot_every

# Begin!
for epoch in range(1, n_epochs + 1):

    # Get training data for this cycle
    training_pair = variables_from_pair(random.choice(pairs))
    input_variable = training_pair[0]
    target_variable = training_pair[1]

    # Run the train function
    loss = train(input_variable, target_variable, encoder, decoder, encoder_optimizer,

    # Keep track of loss
    print_loss_total += loss
    plot_loss_total += loss

    if epoch == 0: continue

    if epoch % print_every == 0:
        print_loss_avg = print_loss_total / print_every
        print_loss_total = 0
        print_summary = '%s (%d %d%%) %.4f' % (time_since(start, epoch / n_epochs), epo
        print(print_summary)
```

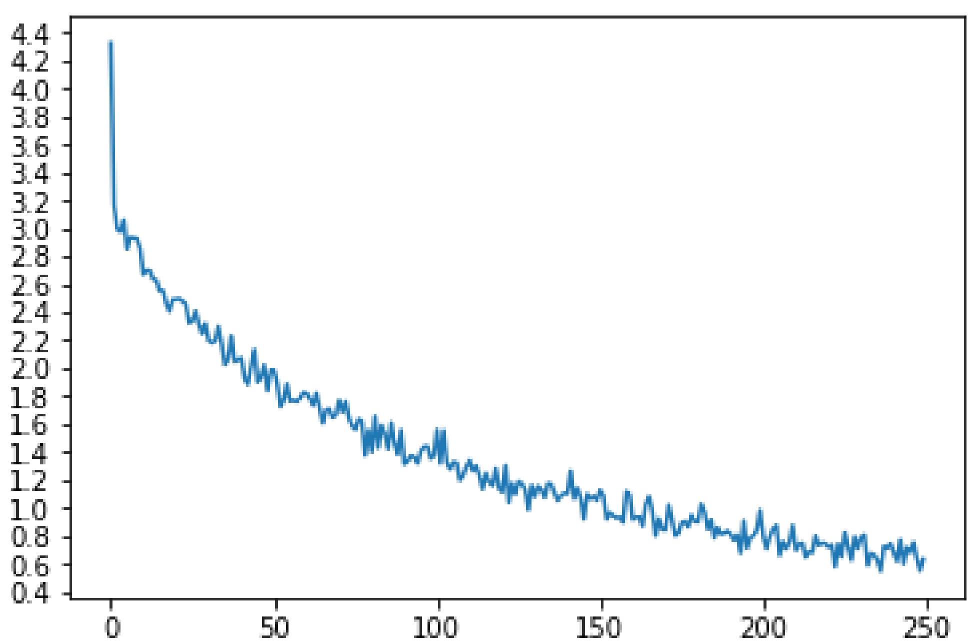
```
            if epoch % plot_every == 0:
                plot_loss_avg = plot_loss_total / plot_every
                plot_losses.append(plot_loss_avg)
                plot_loss_total = 0
```

```
0m 48s (- 39m 42s) (1000 2%) 3.3051
1m 39s (- 39m 46s) (2000 4%) 2.9005
2m 30s (- 39m 14s) (3000 6%) 2.6657
3m 21s (- 38m 41s) (4000 8%) 2.4938
4m 14s (- 38m 6s) (5000 10%) 2.4491
5m 6s (- 37m 25s) (6000 12%) 2.3249
5m 58s (- 36m 43s) (7000 14%) 2.2104
6m 51s (- 35m 58s) (8000 16%) 2.0840
7m 43s (- 35m 9s) (9000 18%) 2.0048
8m 35s (- 34m 22s) (10000 20%) 1.9372
9m 28s (- 33m 34s) (11000 22%) 1.8518
10m 20s (- 32m 43s) (12000 24%) 1.7812
11m 13s (- 31m 55s) (13000 26%) 1.7741
12m 4s (- 31m 3s) (14000 28%) 1.6651
12m 57s (- 30m 13s) (15000 30%) 1.6897
13m 49s (- 29m 23s) (16000 32%) 1.5500
14m 42s (- 28m 33s) (17000 34%) 1.5214
15m 35s (- 27m 43s) (18000 36%) 1.4868
16m 24s (- 26m 46s) (19000 38%) 1.3412
17m 14s (- 25m 52s) (20000 40%) 1.4016
18m 6s (- 25m 0s) (21000 42%) 1.4071
18m 59s (- 24m 10s) (22000 44%) 1.2744
19m 51s (- 23m 19s) (23000 46%) 1.2544
20m 45s (- 22m 29s) (24000 48%) 1.2026
21m 38s (- 21m 38s) (25000 50%) 1.1451
22m 31s (- 20m 47s) (26000 52%) 1.1275
23m 24s (- 19m 56s) (27000 54%) 1.1227
24m 15s (- 19m 3s) (28000 56%) 1.1037
25m 7s (- 18m 11s) (29000 57%) 1.1286
26m 0s (- 17m 20s) (30000 60%) 1.0441
26m 53s (- 16m 28s) (31000 62%) 1.0076
27m 46s (- 15m 37s) (32000 64%) 0.9936
28m 38s (- 14m 45s) (33000 66%) 0.9351
29m 30s (- 13m 53s) (34000 68%) 0.9251
30m 21s (- 13m 0s) (35000 70%) 0.8797
31m 14s (- 12m 9s) (36000 72%) 0.9016
32m 9s (- 11m 17s) (37000 74%) 0.9311
33m 2s (- 10m 26s) (38000 76%) 0.8212
33m 56s (- 9m 34s) (39000 78%) 0.7940
34m 49s (- 8m 42s) (40000 80%) 0.8256
35m 42s (- 7m 50s) (41000 82%) 0.8033
36m 36s (- 6m 58s) (42000 84%) 0.7511
37m 30s (- 6m 6s) (43000 86%) 0.7031
```

```
38m 24s (- 5m 14s) (44000 88%) 0.7407
39m 17s (- 4m 21s) (45000 90%) 0.6846
40m 11s (- 3m 29s) (46000 92%) 0.7335
41m 4s (- 2m 37s) (47000 94%) 0.7039
41m 54s (- 1m 44s) (48000 96%) 0.6708
42m 48s (- 0m 52s) (49000 98%) 0.6763
43m 41s (- 0m 0s) (50000 100%) 0.6513
```

```
In [27]: import matplotlib.pyplot as plt
         import matplotlib.ticker as ticker
         import numpy as np
         %matplotlib inline

         def show_plot(points):
             plt.figure()
             fig, ax = plt.subplots()
             loc = ticker.MultipleLocator(base=0.2) # put ticks at regular intervals
             ax.yaxis.set_major_locator(loc)
             plt.plot(points)

         show_plot(plot_losses)
```

```
<Figure size 432x288 with 0 Axes>
```




```
In [35]: type(EOS_token)
```

Out[35]: int

In [36]:
```
def evaluate(sentence, max_length=MAX_LENGTH):
    input_variable = variable_from_sentence(input_lang, sentence)
    input_length = input_variable.size()[0]

    # Run through encoder
    encoder_hidden = encoder.init_hidden()
    encoder_outputs, encoder_hidden = encoder(input_variable, encoder_hidden)

    # Create starting vectors for decoder
    decoder_input = Variable(torch.LongTensor([[SOS_token]])) # SOS
    decoder_context = Variable(torch.zeros(1, decoder.hidden_size))
    if USE_CUDA:
        decoder_input = decoder_input.cuda()
        decoder_context = decoder_context.cuda()

    decoder_hidden = encoder_hidden

    decoded_words = []
    decoder_attentions = torch.zeros(max_length, max_length)

    # Run through decoder
    for di in range(max_length):
        decoder_output, decoder_context, decoder_hidden, decoder_attention = decoder(de
        decoder_attentions[di,:decoder_attention.size(2)] += decoder_attention.squeeze(

        # Choose top word from output
        topv, topi = decoder_output.data.topk(1)
        ni = topi[0][0].item()
        if ni == EOS_token:
            decoded_words.append('<EOS>')
            break
        else:
            decoded_words.append(output_lang.index2word[ni])

        # Next input is chosen word
        decoder_input = Variable(torch.LongTensor([[ni]]))
        if USE_CUDA: decoder_input = decoder_input.cuda()

    return decoded_words, decoder_attentions[:di+1, :len(encoder_outputs)]
```

In [37]:
```
def evaluate_randomly():
    pair = random.choice(pairs)

    output_words, decoder_attn = evaluate(pair[0])
    output_sentence = ' '.join(output_words)
```

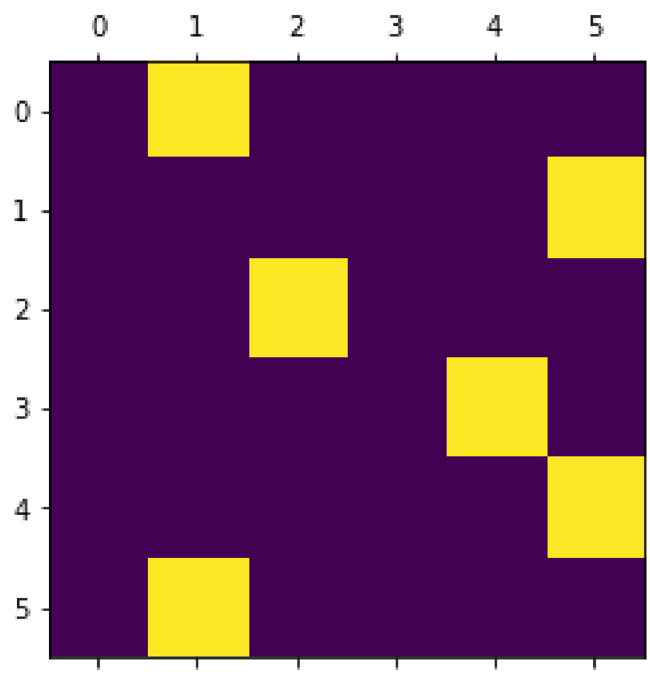
```
        print('>', pair[0])
        print('=', pair[1])
        print('<', output_sentence)
        print('')
```

In [38]:
```
evaluate_randomly()
```

```
> c est un beau jeune homme .
= he is a handsome young man .
< he is a promising young man . <EOS>
```

In [39]:
```
output_words, attentions = evaluate("je suis trop froid .")
plt.matshow(attentions.numpy())
```

Out[39]: <matplotlib.image.AxesImage at 0x7fd8a01592e8>

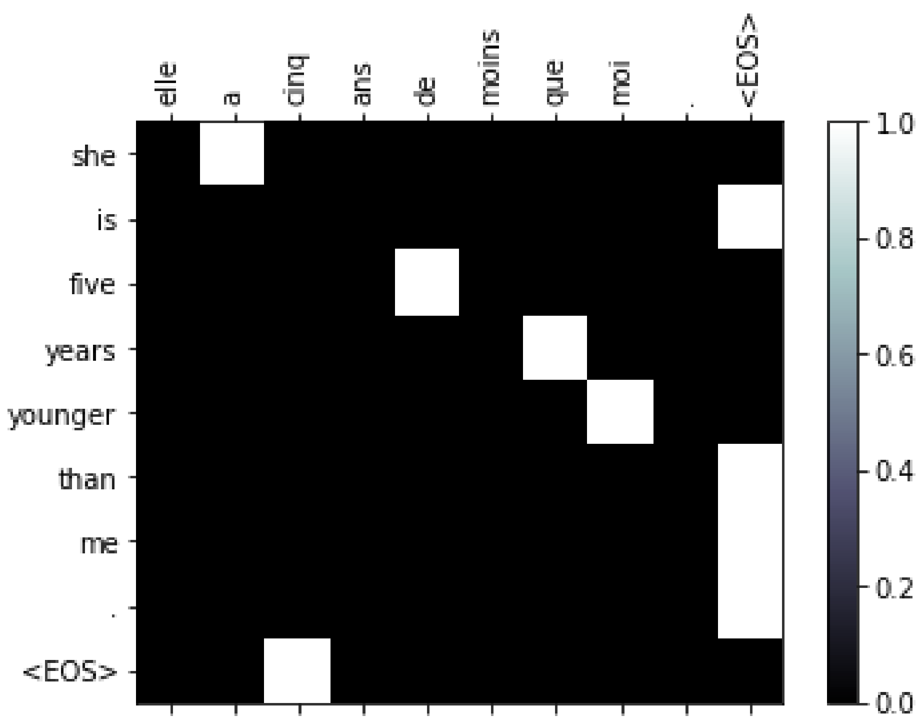


In [40]:
```
def show_attention(input_sentence, output_words, attentions):
    # Set up figure with colorbar
    fig = plt.figure()
    ax = fig.add_subplot(111)
    cax = ax.matshow(attentions.numpy(), cmap='bone')
    fig.colorbar(cax)

    # Set up axes
```

```
        ax.set_xticklabels([''] + input_sentence.split(' ') + ['<EOS>'], rotation=90)
        ax.set_yticklabels([''] + output_words)

        # Show label at every tick
        ax.xaxis.set_major_locator(ticker.MultipleLocator(1))
        ax.yaxis.set_major_locator(ticker.MultipleLocator(1))

        plt.show()
        plt.close()

    def evaluate_and_show_attention(input_sentence):
        output_words, attentions = evaluate(input_sentence)
        print('input =', input_sentence)
        print('output =', ' '.join(output_words))
        show_attention(input_sentence, output_words, attentions)
```

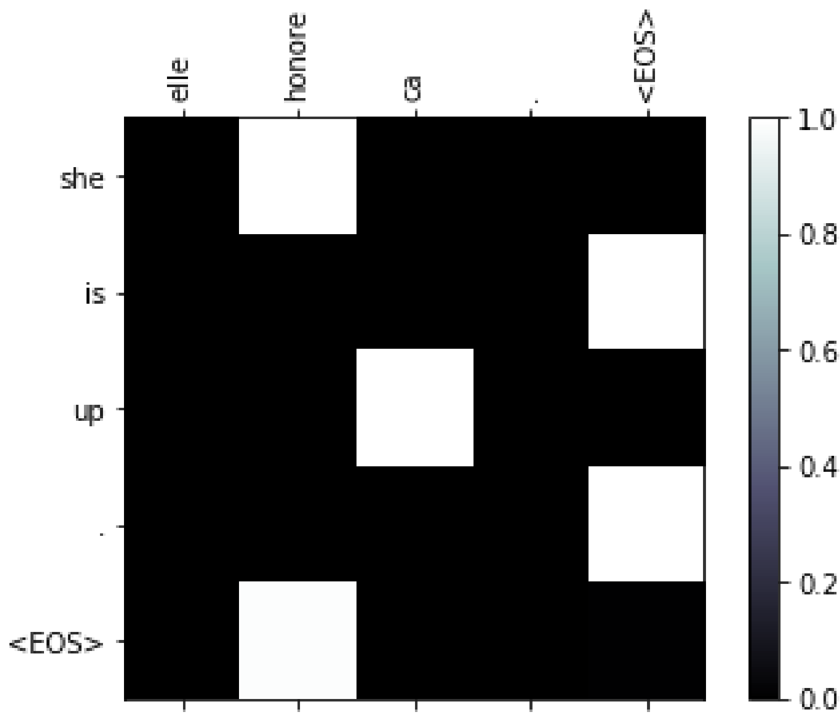
```
In [41]: evaluate_and_show_attention("elle a cinq ans de moins que moi .")
```

```
input = elle a cinq ans de moins que moi .
output = she is five years younger than me . <EOS>
```




```
In [70]: evaluate_and_show_attention("elle honore ca .")
```

```
input = elle honore ca .
output = she is up . <EOS>
```




```
In [ ]:
```