**Adapted from PyTorch tutorial https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial (https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutoria originally accessed on 03-28-2023**

In [1]:
```
# For tips on running notebooks in Google Colab, see
# https://pytorch.org/tutorials/beginner/colab
%matplotlib inline
```

## Preliminary: Torch embedding layers

Embedding layers take torch tensors of int type (either int32 or int64/long) and return vectors for each int. Essentially, they are "learnable" lookup tables. They are equivalent to creating one-hot vectors and then multiplying by a learnable embedding matrix. https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html (https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html)

If $W \in \mathbb{R}^{V \times H}$ represents an embedding matrix where $H$ is the embedding dimension and $V$ is the vocabulary size, then we can define the embedding for index $0 \leq i < V$ as follows:

$$\text{embedding}(i) = w_i = W^T \text{OneHot(i)}$$

The hidden size is appended to the shape of the indices input. For example, if the input indices had shape (4,3,2) and the hidden size was 100, then the output shape would be (4,3,2,100)

In [2]:
```
import torch
import torch.nn as nn
vocab_size = 5
embedding_size = 2
embedding = nn.Embedding(vocab_size, embedding_size)
input_idx = torch.tensor([1,2,1])
input_embed = embedding(input_idx)
print('Input indices')
print(input_idx.unsqueeze(0).T)
print('After embedding (notice that has grad_fn since embeddings are lea
print(input_embed)

print('\nThis is equivalent to one hot encoding and then matrix multipli
def one_hot(idx, vocab_size):
    out = torch.zeros((*idx.size(), vocab_size))
    for i, j in enumerate(idx):
        out[i, j] = 1
    return out
print('One hot encoding')
print(one_hot(input_idx, vocab_size))
print('W.T times one-hot encodings (batched) and then converted to row v
print((embedding.weight.T @ one_hot(input_idx, vocab_size).T).T)
```

```
Input indices
tensor([[1],
        [2],
        [1]])
After embedding (notice that has grad_fn since embeddings are learnab
le)
tensor([[-1.8059, -0.1946],
        [ 0.3467,  0.9974],
        [-1.8059, -0.1946]], grad_fn=<EmbeddingBackward>)

This is equivalent to one hot encoding and then matrix multiplication
One hot encoding
tensor([[0., 1., 0., 0., 0.],
        [0., 0., 1., 0., 0.],
        [0., 1., 0., 0., 0.]])
W.T times one-hot encodings (batched) and then converted to row vecto
rs
tensor([[-1.8059, -0.1946],
        [ 0.3467,  0.9974],
        [-1.8059, -0.1946]], grad_fn=<PermuteBackward>)
```

# NLP From Scratch: Translation with a Sequence to Sequence Network and Attention

**Author**: [Sean Robertson (https://github.com/spro/practical-pytorch)](https://github.com/spro/practical-pytorch)

This is the third and final tutorial on doing "NLP From Scratch", where we write our own classes and functions to preprocess the data to do our NLP modeling tasks. We hope after you complete this tutorial that you'll proceed to learn how `torchtext` can handle much of this preprocessing for you in the three tutorials immediately following this one.

In this project we will be teaching a neural network to translate from French to English.

::

```
[KEY: > input, = target, < output]

> il est en train de peindre un tableau .
= he is painting a picture .
< he is painting a picture .

> pourquoi ne pas essayer ce vin delicieux ?
= why not try that delicious wine ?
< why not try that delicious wine ?

> elle n est pas poete mais romanciere .
= she is not a poet but a novelist .
< she not not a poet but a novelist .

> vous etes trop maigre .
= you re too skinny .
< you re all alone .
```

... to varying degrees of success.

This is made possible by the simple but powerful idea of the [sequence to sequence network (https://arxiv.org/abs/1409.3215)](https://arxiv.org/abs/1409.3215)_, in which two recurrent neural networks work together to transform one sequence to another. An encoder network condenses an input sequence into a vector, and a decoder network unfolds that vector into a new sequence.

.. figure:: /_static/img/seq-seq-images/seq2seq.png :alt:

To improve upon this model we'll use an [attention mechanism (https://arxiv.org/abs/1409.0473)](https://arxiv.org/abs/1409.0473)_, which lets the decoder learn to focus over a specific range of the input sequence.

**Recommended Reading:**

I assume you have at least installed PyTorch, know Python, and understand Tensors:

- [https://pytorch.org/ (https://pytorch.org/)](https://pytorch.org/) For installation instructions
- :doc: `/beginner/deep_learning_60min_blitz` to get started with PyTorch in general
- :doc: `/beginner/pytorch_with_examples` for a wide and deep overview
- :doc: `/beginner/former_torchies_tutorial` if you are former Lua Torch user

It would also be useful to know about Sequence to Sequence networks and how they work:

- [Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation (https://arxiv.org/abs/1406.1078)](https://arxiv.org/abs/1406.1078)_
- [Sequence to Sequence Learning with Neural Networks (https://arxiv.org/abs/1409.3215)](https://arxiv.org/abs/1409.3215)_
- [Neural Machine Translation by Jointly Learning to Align and Translate (https://arxiv.org/abs/1409.0473)](https://arxiv.org/abs/1409.0473)_

- [A Neural Conversational Model (https://arxiv.org/abs/1506.05869)_](https://arxiv.org/abs/1506.05869)

You will also find the previous tutorials on :doc: `/intermediate/char_rnn_classification_tutorial` and :doc: `/intermediate/char_rnn_generation_tutorial` helpful as those concepts are very similar to the Encoder and Decoder models, respectively.

**Requirements**

In [3]:
```
from __future__ import unicode_literals, print_function, division
from io import open
import unicodedata
import string
import re
import random

import torch
import torch.nn as nn
from torch import optim
import torch.nn.functional as F

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

# Loading data files

The data for this project is a set of many thousands of English to French translation pairs.

[This question on Open Data Stack Exchange (https://opendata.stackexchange.com/questions/3888/dataset-of-sentences-translated-into-many-languages)_](https://opendata.stackexchange.com/questions/3888/dataset-of-sentences-translated-into-many-languages) pointed me to the open translation site [https://tatoeba.org/ (https://tatoeba.org/)](https://tatoeba.org/) which has downloads available at [https://tatoeba.org/eng/downloads (https://tatoeba.org/eng/downloads)](https://tatoeba.org/eng/downloads) - and better yet, someone did the extra work of splitting language pairs into individual text files here: [https://www.manythings.org/anki/ (https://www.manythings.org/anki/)](https://www.manythings.org/anki/)

The English to French pairs are too big to include in the repo, so download to `data/eng-fra.txt` before continuing. The file is a tab separated list of translation pairs:

::

```
I am cold.    J'ai froid.
```

.. Note:: Download the data from [here (https://download.pytorch.org/tutorial/data.zip)](https://download.pytorch.org/tutorial/data.zip) and extract it to the current directory.

Similar to the character encoding used in the character-level RNN tutorials, we will be representing each word in a language as a one-hot vector, or giant vector of zeros except for a single one (at the index of the word). Compared to the dozens of characters that might exist in a language, there are many many more words, so the encoding vector is much larger. We will however cheat a bit and trim the data to only use a few thousand words per language.

.. figure:: /_static/img/seq-seq-images/word-encoding.png :alt:

We'll need a unique index per word to use as the inputs and targets of the networks later. To keep track of all this we will use a helper class called `Lang` which has word → index ( `word2index` ) and index → word ( `index2word` ) dictionaries, as well as a count of each word `word2count` which will be used to replace rare words later.

In [4]:
```
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 addSentence(self, sentence):
        for word in sentence.split(' '):
            self.addWord(word)

    def addWord(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
```

The files are all in Unicode, to simplify we will turn Unicode characters to ASCII, make everything lowercase, and trim most punctuation.

In [5]:
```
# Turn a Unicode string to plain ASCII, thanks to
# https://stackoverflow.com/a/518232/2809427
def unicodeToAscii(s):
    return ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn'
    )

# Lowercase, trim, and remove non-letter characters


def normalizeString(s):
    s = unicodeToAscii(s.lower().strip())
    s = re.sub(r"([.!?])", r" \1", s)
    s = re.sub(r"[^a-zA-Z.!?]+", r" ", s)
    return s
```

To read the data file we will split the file into lines, and then split lines into pairs. The files are all English → Other Language, so if we want to translate from Other Language → English I added the `reverse` flag to reverse the pairs.

In [6]:
```
def readLangs(lang1, lang2, reverse=False):
    print("Reading lines...")

    # Read the file and split into lines
    lines = open('data/%s-%s.txt' % (lang1, lang2), encoding='utf-8').\
        read().strip().split('\n')

    # Split every line into pairs and normalize
    pairs = [[normalizeString(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
```

Since there are a *lot* of example sentences and we want to train something quickly, we'll trim the data set to only relatively short and simple sentences. Here the maximum length is 10 words (that includes ending punctuation) and we're filtering to sentences that translate to the form "I am" or "He is" etc. (accounting for apostrophes replaced earlier).

In [7]:
```python
MAX_LENGTH = 10

eng_prefixes = (
    "i am ", "i m ",
    "he is", "he s ",
    "she is", "she s ",
    "you are", "you re ",
    "we are", "we re ",
    "they are", "they re "
)


def filterPair(p):
    return len(p[0].split(' ')) < MAX_LENGTH and \
        len(p[1].split(' ')) < MAX_LENGTH and \
        p[1].startswith(eng_prefixes)


def filterPairs(pairs):
    return [pair for pair in pairs if filterPair(pair)]
```

The full process for preparing the data is:

- Read text file and split into lines, split lines into pairs
- Normalize text, filter by length and content
- Make word lists from sentences in pairs

In [8]:
```
def prepareData(lang1, lang2, reverse=False):
    input_lang, output_lang, pairs = readLangs(lang1, lang2, reverse)
    print("Read %s sentence pairs" % len(pairs))
    pairs = filterPairs(pairs)
    print("Trimmed to %s sentence pairs" % len(pairs))
    print("Counting words...")
    for pair in pairs:
        input_lang.addSentence(pair[0])
        output_lang.addSentence(pair[1])
    print("Counted words:")
    print(input_lang.name, input_lang.n_words)
    print(output_lang.name, output_lang.n_words)
    return input_lang, output_lang, pairs


input_lang, output_lang, pairs = prepareData('eng', 'fra', True)
print(random.choice(pairs))
```

```
Reading lines...
Read 135842 sentence pairs
Trimmed to 10599 sentence pairs
Counting words...
Counted words:
fra 4345
eng 2803
['je vous suis reconnaissant de votre gentillesse .', 'i am grateful
to you for your kindness .']
```

## The Seq2Seq Model

A Recurrent Neural Network, or RNN, is a network that operates on a sequence and uses its own output as input for subsequent steps.

A [Sequence to Sequence network (https://arxiv.org/abs/1409.3215)](https://arxiv.org/abs/1409.3215)*, or seq2seq network, or* *[Encoder Decoder network (https://arxiv.org/pdf/1406.1078v3.pdf)](https://arxiv.org/pdf/1406.1078v3.pdf)*, is a model consisting of two RNNs called the encoder and decoder. The encoder reads an input sequence and outputs a single vector, and the decoder reads that vector to produce an output sequence.

.. figure:: /_static/img/seq-seq-images/seq2seq.png :alt:

Unlike sequence prediction with a single RNN, where every input corresponds to an output, the seq2seq model frees us from sequence length and order, which makes it ideal for translation between two languages.

Consider the sentence "Je ne suis pas le chat noir" → "I am not the black cat". Most of the words in the input sentence have a direct translation in the output sentence, but are in slightly different orders, e.g. "chat noir" and "black cat". Because of the "ne/pas" construction there is also one more word in the input sentence. It would be difficult to produce a correct translation directly from the sequence of input words.

With a seq2seq model the encoder creates a single vector which, in the ideal case, encodes

## The Encoder

The encoder of a seq2seq network is a RNN that outputs some value for every word from the input sentence. For every input word the encoder outputs a vector and a hidden state, and uses the hidden state for the next input word.

.. figure:: /_static/img/seq-seq-images/encoder-network.png :alt:

In [9]:
```
class EncoderRNN(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(EncoderRNN, self).__init__()
        self.hidden_size = hidden_size

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

    def forward(self, input, hidden):
        embedded = self.embedding(input).view(1, 1, -1)
        output = embedded
        output, hidden = self.gru(output, hidden)
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size, device=device)
```

## The Decoder

The decoder is another RNN that takes the encoder output vector(s) and outputs a sequence of words to create the translation.

### Simple Decoder

In the simplest seq2seq decoder we use only last output of the encoder. This last output is sometimes called the *context vector* as it encodes context from the entire sequence. This context vector is used as the initial hidden state of the decoder.

At every step of decoding, the decoder is given an input token and hidden state. The initial input token is the start-of-string `<SOS>` token, and the first hidden state is the context vector (the encoder's last hidden state).

.. figure:: /_static/img/seq-seq-images/decoder-network.png :alt:

In [10]:
```
class DecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size):
        super(DecoderRNN, self).__init__()
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(output_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size)
        self.out = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        # `input` is a single long (int64) value (representing a single
        # `hidden` is (1, 1, 256) (L, N, H) (1 word, 1 sample in batch,
        output = self.embedding(input).view(1, 1, -1)
        output = F.relu(output)
        output, hidden = self.gru(output, hidden)

        # Convert to output log probability (of different size)
        #  (1, 4345) (N, V) where V is vocab size of output
        output = self.softmax(self.out(output[0]))
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size, device=device)
```

I encourage you to train and observe the results of this model, but to save space we'll be going straight for the gold and introducing the Attention Mechanism.

#### Attention Decoder

If only the context vector is passed between the encoder and decoder, that single vector carries the burden of encoding the entire sentence.

Attention allows the decoder network to "focus" on a different part of the encoder's outputs for every step of the decoder's own outputs. First we calculate a set of *attention weights*. These will be multiplied by the encoder output vectors to create a weighted combination. The result (called `attn_applied` in the code) should contain information about that specific part of the input sequence, and thus help the decoder choose the right output words.

.. figure:: [https://i.imgur.com/1152PYf.png (https://i.imgur.com/1152PYf.png)](https://i.imgur.com/1152PYf.png) :alt:

Calculating the attention weights is done with another feed-forward layer `attn` , using the decoder's input and hidden state as inputs. Because there are sentences of all sizes in the training data, to actually create and train this layer we have to choose a maximum sentence length (input length, for encoder outputs) that it can apply to. Sentences of the maximum length will use all the attention weights, while shorter sentences will only use the first few.

.. figure:: /_static/img/seq-seq-images/attention-decoder-network.png :alt:

In [11]:
```
class AttnDecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size, dropout_p=0.1, max_leng
        super(AttnDecoderRNN, self).__init__()
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.dropout_p = dropout_p
        self.max_length = max_length

        self.embedding = nn.Embedding(self.output_size, self.hidden_size
        self.attn = nn.Linear(self.hidden_size * 2, self.max_length)
        self.attn_combine = nn.Linear(self.hidden_size * 2, self.hidden_
        self.dropout = nn.Dropout(self.dropout_p)
        self.gru = nn.GRU(self.hidden_size, self.hidden_size)
        self.out = nn.Linear(self.hidden_size, self.output_size)

    def forward(self, input, hidden, encoder_outputs):
        # `input` is a single long (int64) value (representing a single
        # `hidden` is (1, 1, 256) (L, N, H) (1 word, 1 sample in batch,
        # `encoder_outputs` (10, 256) (L_in, H) (total length of input,

        # Get input embedding
        # (1, 1, 256) (L, N, H)
        embedded = self.embedding(input).view(1, 1, -1)
        embedded = self.dropout(embedded) # Add random 0s

        # Compute attention weights
        # (1, 10) (N, L_max)
        attn_weights = F.softmax(
            self.attn(torch.cat((embedded[0], hidden[0]), 1)), dim=1)

        # Apply attention to get weighted sum of encoder outputs
        # (1,1,256) = (1,1,10) x (1,10,256) = (L,N,L_max) x (L,L_max,H)
        attn_applied = torch.bmm(attn_weights.unsqueeze(0),
                                 encoder_outputs.unsqueeze(0))

        # Apply another fully connected layer after attention for more p
        # (1,512) (N, 2*H)
        output = torch.cat((embedded[0], attn_applied[0]), 1)
        # (1,256) -> (1,1,256) (L, N, H)
        output = self.attn_combine(output).unsqueeze(0)
        # (1,1,256)
        output = F.relu(output)

        # Apply standard RNN cell
        # (1,1,256), (1,1,256)
        output, hidden = self.gru(output, hidden)

        # Convert to output log probability (of different size)
        #  (1, 4345) (N, V) where V is vocab size of output
        output = F.log_softmax(self.out(output[0]), dim=1)
        return output, hidden, attn_weights

    def initHidden(self):
```

```
        return torch.zeros(1, 1, self.hidden_size, device=device)
```

**Note**

There are other forms of attention that work around the length limitation by using a relative position approach. Read about "local attention" in [Effective Approaches to Attention-based Neural Machine Translation](https://arxiv.org/abs/1508.04025)_.

# Training

## Preparing Training Data

To train, for each pair we will need an input tensor (indexes of the words in the input sentence) and target tensor (indexes of the words in the target sentence). While creating these vectors we will append the EOS token to both sequences.

In [12]:
```
def indexesFromSentence(lang, sentence):
    return [lang.word2index[word] for word in sentence.split(' ')]


def tensorFromSentence(lang, sentence):
    indexes = indexesFromSentence(lang, sentence)
    indexes.append(EOS_token)
    return torch.tensor(indexes, dtype=torch.long, device=device).view(-


def tensorsFromPair(pair):
    input_tensor = tensorFromSentence(input_lang, pair[0])
    target_tensor = tensorFromSentence(output_lang, pair[1])
    return (input_tensor, target_tensor)
```

## Training the Model

To train we run the input sentence through the encoder, and keep track of every output and the latest hidden state. Then the decoder is given the `<SOS>` token as its first input, and the last hidden state of the encoder as its first hidden state.

"Teacher forcing" is the concept of using the real target outputs as each next input, instead of using the decoder's guess as the next input. Using teacher forcing causes it to converge faster but [when the trained network is exploited, it may exhibit instability (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.378.4095&rep=rep1&type=pdf)](http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.378.4095&rep=rep1&type=pdf)_.

You can observe outputs of teacher-forced networks that read with coherent grammar but wander far from the correct translation - intuitively it has learned to represent the output grammar and can "pick up" the meaning once the teacher tells it the first few words, but it has not properly learned how to create the sentence from the translation in the first place.

Because of the freedom PyTorch's autograd gives us, we can randomly choose to use teacher forcing or not with a simple if statement. Turn `teacher_forcing_ratio` up to use more of it.

```
In [13]: teacher_forcing_ratio = 0.5


def train(input_tensor, target_tensor, encoder, decoder, encoder_optimi
    encoder_hidden = encoder.initHidden()

    encoder_optimizer.zero_grad()
    decoder_optimizer.zero_grad()

    input_length = input_tensor.size(0)
    target_length = target_tensor.size(0)

    # Need to initialize up to max_length for attention
    encoder_outputs = torch.zeros(max_length, encoder.hidden_size, devic

    loss = 0

    # Loop through input with encoder saving outputs
    #  for use in decoder step
    for ei in range(input_length):
        encoder_output, encoder_hidden = encoder(
            input_tensor[ei], encoder_hidden)
        encoder_outputs[ei] = encoder_output[0, 0]

    # Initialize decoder input with start of sentence SOS token
    decoder_input = torch.tensor([[SOS_token]], device=device)

    # Copy the last encoder_hidden value to initialize the decoder_hidde
    decoder_hidden = encoder_hidden

    # Randomly choose to use teacher forcing or not
    use_teacher_forcing = True if random.random() < teacher_forcing_rat

    # Run decoder RNN and compute loss for each term
    if use_teacher_forcing:
        # Teacher forcing: Feed the target as the next input
        for di in range(target_length):
            decoder_output, decoder_hidden, decoder_attention = decoder
                decoder_input, decoder_hidden, encoder_outputs)
            loss += criterion(decoder_output, target_tensor[di])
            decoder_input = target_tensor[di]  # Teacher forcing

    else:
        # Without teacher forcing: use its own predictions as the next 
        for di in range(target_length):
            decoder_output, decoder_hidden, decoder_attention = decoder
                decoder_input, decoder_hidden, encoder_outputs)
            topv, topi = decoder_output.topk(1)
            decoder_input = topi.squeeze().detach()  # detach from histo

            loss += criterion(decoder_output, target_tensor[di])
            # Stop of EOS predicted
            if decoder_input.item() == EOS_token:
                break

    loss.backward()
```

```
    encoder_optimizer.step()
    decoder_optimizer.step()

    return loss.item() / target_length
```

This is a helper function to print time elapsed and estimated time remaining given the current time and progress %.

In [14]:
```
import time
import math


def asMinutes(s):
    m = math.floor(s / 60)
    s -= m * 60
    return '%dm %ds' % (m, s)


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

The whole training process looks like this:

- Start a timer
- Initialize optimizers and criterion
- Create set of training pairs
- Start empty losses array for plotting

Then we call `train` many times and occasionally print the progress (% of examples, time so far, estimated time) and average loss.

In [15]:

```
def trainIters(encoder, decoder, n_iters, print_every=1000, plot_every=1
    start = time.time()
    plot_losses = []
    print_loss_total = 0  # Reset every print_every
    plot_loss_total = 0  # Reset every plot_every

    encoder_optimizer = optim.SGD(encoder.parameters(), lr=learning_rate
    decoder_optimizer = optim.SGD(decoder.parameters(), lr=learning_rate
    training_pairs = [tensorsFromPair(random.choice(pairs))
                      for i in range(n_iters)]
    criterion = nn.NLLLoss()

    for iter in range(1, n_iters + 1):
        training_pair = training_pairs[iter - 1]
        input_tensor = training_pair[0]
        target_tensor = training_pair[1]

        loss = train(input_tensor, target_tensor, encoder,
                     decoder, encoder_optimizer, decoder_optimizer, crit
        print_loss_total += loss
        plot_loss_total += loss

        if iter % print_every == 0:
            print_loss_avg = print_loss_total / print_every
            print_loss_total = 0
            print('%s (%d %d%%) %.4f' % (timeSince(start, iter / n_iters
                                         iter, iter / n_iters * 100, pri

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

    showPlot(plot_losses)
```

## Plotting results

Plotting is done with matplotlib, using the array of loss values `plot_losses` saved while training.

In [16]:

```
import matplotlib.pyplot as plt
plt.switch_backend('agg')
import matplotlib.ticker as ticker
import numpy as np


def showPlot(points):
    plt.figure()
    fig, ax = plt.subplots()
    # this locator puts ticks at regular intervals
    loc = ticker.MultipleLocator(base=0.2)
    ax.yaxis.set_major_locator(loc)
    plt.plot(points)
```

## Evaluation

Evaluation is mostly the same as training, but there are no targets so we simply feed the decoder's predictions back to itself for each step. Every time it predicts a word we add it to the output string, and if it predicts the EOS token we stop there. We also store the decoder's attention outputs for display later.

In [17]:
```
def evaluate(encoder, decoder, sentence, max_length=MAX_LENGTH):
    with torch.no_grad():
        input_tensor = tensorFromSentence(input_lang, sentence)
        input_length = input_tensor.size()[0]
        encoder_hidden = encoder.initHidden()

        encoder_outputs = torch.zeros(max_length, encoder.hidden_size, d

        for ei in range(input_length):
            encoder_output, encoder_hidden = encoder(input_tensor[ei],
                                                     encoder_hidden)
            encoder_outputs[ei] += encoder_output[0, 0]

        decoder_input = torch.tensor([[SOS_token]], device=device)  # S

        decoder_hidden = encoder_hidden

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

        for di in range(max_length):
            decoder_output, decoder_hidden, decoder_attention = decoder
                decoder_input, decoder_hidden, encoder_outputs)
            decoder_attentions[di] = decoder_attention.data
            topv, topi = decoder_output.data.topk(1)
            if topi.item() == EOS_token:
                decoded_words.append('<EOS>')
                break
            else:
                decoded_words.append(output_lang.index2word[topi.item()

            decoder_input = topi.squeeze().detach()

        return decoded_words, decoder_attentions[:di + 1]
```

We can evaluate random sentences from the training set and print out the input, target, and output to make some subjective quality judgements:

In [18]:
```
def evaluateRandomly(encoder, decoder, n=10):
    for i in range(n):
        pair = random.choice(pairs)
        print('>', pair[0])
        print('=', pair[1])
        output_words, attentions = evaluate(encoder, decoder, pair[0])
        output_sentence = ' '.join(output_words)
        print('<', output_sentence)
        print('')
```

## Training and Evaluating

With all these helper functions in place (it looks like extra work, but it makes it easier to run multiple experiments) we can actually initialize a network and start training.

Remember that the input sentences were heavily filtered. For this small dataset we can use relatively small networks of 256 hidden nodes and a single GRU layer. After about 40 minutes on a MacBook CPU we'll get some reasonable results.

.. Note:: If you run this notebook you can train, interrupt the kernel, evaluate, and continue training later. Comment out the lines where the encoder and decoder are initialized and run `trainIters` again.

In [19]:

```
hidden_size = 256
encoder1 = EncoderRNN(input_lang.n_words, hidden_size).to(device)
attn_decoder1 = AttnDecoderRNN(hidden_size, output_lang.n_words, dropout

trainIters(encoder1, attn_decoder1, 75000, print_every=5000)
```

```
2m 30s (- 35m 3s) (5000 6%) 2.8297
4m 59s (- 32m 27s) (10000 13%) 2.2812
7m 31s (- 30m 4s) (15000 20%) 1.9918
10m 2s (- 27m 37s) (20000 26%) 1.7500
12m 32s (- 25m 4s) (25000 33%) 1.5230
15m 1s (- 22m 32s) (30000 40%) 1.3963
17m 32s (- 20m 3s) (35000 46%) 1.2212
20m 5s (- 17m 34s) (40000 53%) 1.1255
22m 41s (- 15m 7s) (45000 60%) 0.9851
25m 11s (- 12m 35s) (50000 66%) 0.9019
27m 43s (- 10m 4s) (55000 73%) 0.8400
30m 20s (- 7m 35s) (60000 80%) 0.7350
33m 11s (- 5m 6s) (65000 86%) 0.6786
35m 57s (- 2m 34s) (70000 93%) 0.6633
38m 40s (- 0m 0s) (75000 100%) 0.5992
```

In [20]:
```
evaluateRandomly(encoder1, attn_decoder1)
```

```
> je suis si epuisee !
= i am so exhausted !
< i m so stupid . <EOS>

> tu es bon .
= you are good .
< you re good . <EOS>

> vous etes tres solitaire .
= you re very lonely .
< you re very lonely . <EOS>

> nous ne sommes pas ouvertes .
= we re not open .
< we re not open . <EOS>

> il est a court de fonds .
= he is running short of funds .
< he s running of of of . <EOS>

> il est handicape mental .
= he is mentally handicapped .
< he is rather good . <EOS>

> je suis tres fiere de mon pere .
= i m very proud of my father .
< i m very proud of my father . <EOS>

> tu ne cooperes pas .
= you re not cooperating .
< you re not cooperating . <EOS>

> il est frais emoulu de l universite .
= he is fresh from college .
< he is fresh out of college . <EOS>

> je suis ravie de vous aider .
= i am glad to help you .
< i am glad to help you . <EOS>
```

## Visualizing Attention

A useful property of the attention mechanism is its highly interpretable outputs. Because it is used to weight specific encoder outputs of the input sequence, we can imagine looking where the network is focused most at each time step.

You could simply run `plt.matshow(attentions)` to see attention output displayed as a matrix, with the columns being input steps and rows being output steps:

In [21]:
```
output_words, attentions = evaluate(
    encoder1, attn_decoder1, "je suis trop froid .")
plt.matshow(attentions.numpy())
```

Out[21]:
```
<matplotlib.image.AxesImage at 0x7fadcd44f700>
```

For a better viewing experience we will do the extra work of adding axes and labels:

In [22]:
```python
def showAttention(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()


def evaluateAndShowAttention(input_sentence):
    output_words, attentions = evaluate(
        encoder1, attn_decoder1, input_sentence)
    print('input =', input_sentence)
    print('output =', ' '.join(output_words))
    showAttention(input_sentence, output_words, attentions)


evaluateAndShowAttention("elle a cinq ans de moins que moi .")

evaluateAndShowAttention("elle est trop petit .")

evaluateAndShowAttention("je ne crains pas de mourir .")

evaluateAndShowAttention("c est un jeune directeur plein de talent .")
```

```
input = elle a cinq ans de moins que moi .
output = she s two years younger than me . <EOS>
input = elle est trop petit .
output = she is too short . <EOS>
input = je ne crains pas de mourir .
output = i m not scared of dying . <EOS>
input = c est un jeune directeur plein de talent .
output = he s a talented young director . <EOS>

<ipython-input-22-9b51ff8fc28e>:9: UserWarning: FixedFormatter should
only be used together with FixedLocator
  ax.set_xticklabels([''] + input_sentence.split(' ') +
<ipython-input-22-9b51ff8fc28e>:11: UserWarning: FixedFormatter shoul
d only be used together with FixedLocator
  ax.set_yticklabels([''] + output_words)
<ipython-input-22-9b51ff8fc28e>:17: UserWarning: Matplotlib is curren
tly using agg, which is a non-GUI backend, so cannot show the figure.
  plt.show()
```

## Exercises

- Try with a different dataset
  - Another language pair
  - Human → Machine (e.g. IOT commands)
  - Chat → Response
  - Question → Answer
- Replace the embeddings with pre-trained word embeddings such as word2vec or GloVe
- Try with more layers, more hidden units, and more sentences. Compare the training time and results.
- If you use a translation file where pairs have two of the same phrase ( `I am test \t I am test` ), you can use this as an autoencoder. Try this:
  - Train as an autoencoder
  - Save only the Encoder network
  - Train a new Decoder for translation from there