```
%matplotlib inline
```

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

Author: Sean Robertson <https://github.com/spro/practical-pytorch>_ [PyTorch Source](#)

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](#), 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](#), 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/ For installation instructions
- Deep Learning with PyTorch: A 60 Minute Blitz to get started with PyTorch in general
- Learning PyTorch with Examples for a wide and deep overview

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
- Sequence to Sequence Learning with Neural Networks
- Neural Machine Translation by Jointly Learning to Align and Translate
- A Neural Conversational Model
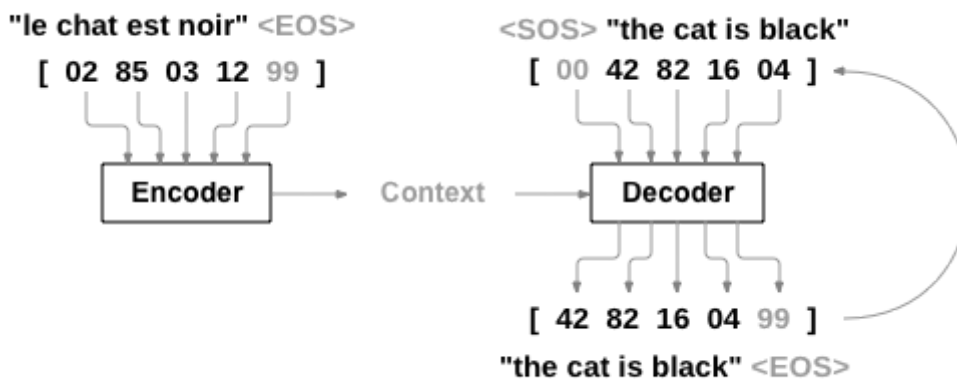
Requirements

```python
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")


device
```

```
device(type='cuda')
```

▾ 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](#) pointed me/him to the open translation site https://tatoeba.org/ which has downloads available at 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/

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>`_ and extract it to the current directory.

```
!wget https://download.pytorch.org/tutorial/data.zip

!unzip data.zip
```

```
--2021-07-11 12:25:54--  https://download.pytorch.org/tutorial/data.zip
Resolving download.pytorch.org (download.pytorch.org)... 99.84.216.2, 99.84.2
Connecting to download.pytorch.org (download.pytorch.org)|99.84.216.2|:443...
HTTP request sent, awaiting response... 200 OK
Length: 2882130 (2.7M) [application/zip]
Saving to: 'data.zip'

data.zip            100%[===================>]   2.75M  --.-KB/s    in 0.04s

2021-07-11 12:25:54 (76.5 MB/s) - 'data.zip' saved [2882130/2882130]

Archive:  data.zip
   creating: data/
  inflating: data/eng-fra.txt
   creating: data/names/
  inflating: data/names/Arabic.txt
  inflating: data/names/Chinese.txt
  inflating: data/names/Czech.txt
  inflating: data/names/Dutch.txt
  inflating: data/names/English.txt
  inflating: data/names/French.txt
  inflating: data/names/German.txt
  inflating: data/names/Greek.txt
  inflating: data/names/Irish.txt
  inflating: data/names/Italian.txt
  inflating: data/names/Japanese.txt
  inflating: data/names/Korean.txt
  inflating: data/names/Polish.txt
  inflating: data/names/Portuguese.txt
  inflating: data/names/Russian.txt
  inflating: data/names/Scottish.txt
  inflating: data/names/Spanish.txt
  inflating: data/names/Vietnamese.txt
```

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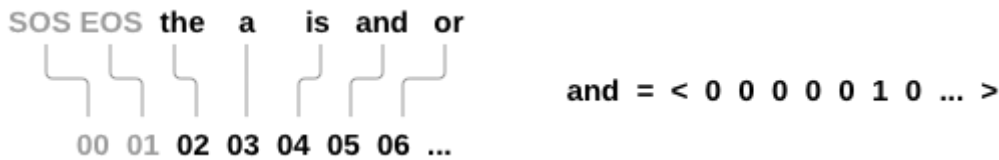
We'll need a unique index per word to use as the inputs and targets of the networks later. To keep track of all this we will use a helper class called `Lang` which has word → index (`word2index`) and index → word (`index2word`) dictionaries, as well as a count of each word `word2count` which will be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

```
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.

```
# 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.

```
    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).

```
    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 "
    )
```

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

```python
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

```python
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
```

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

```
    Reading lines...
    Read 135842 sentence pairs
    Trimmed to 10599 sentence pairs
    Counting words...
    Counted words:
    eng 2803
    fra 4345
    ['i m intrigued .', 'je suis intriguee .']
```
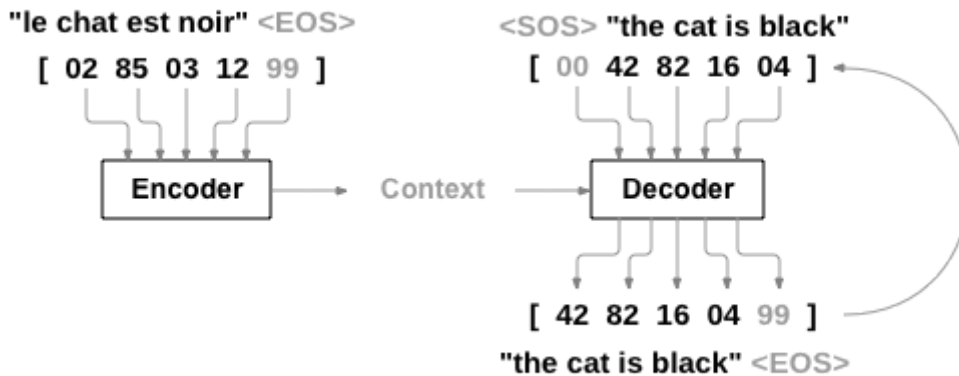
▾ The Seq2Seq Model

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](#), or seq2seq network, or [Encoder Decoder network](#), 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.


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 "meaning" of the input sequence into a single vector — a single point in some N dimensional space of sentences.

▾ 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.
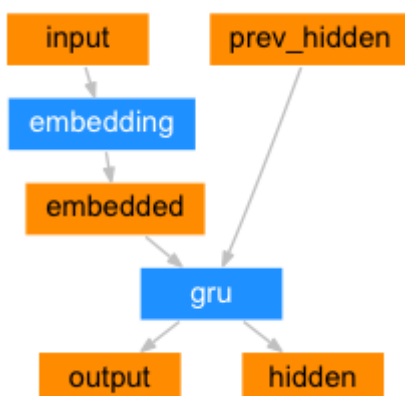
```python
class EncoderRNN(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(EncoderRNN, self).__init__()
        self.hidden_size = hidden_size

        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)
```

▾ 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).
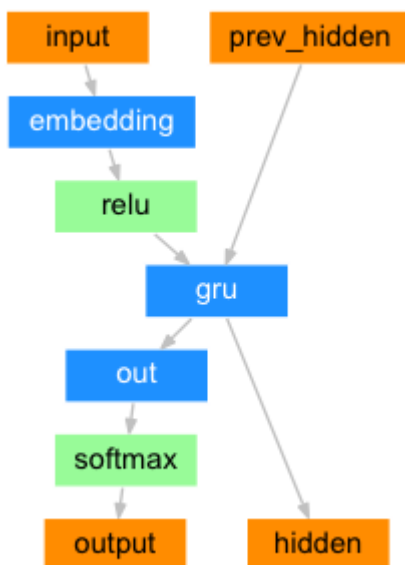

```python
class DecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size):
        super(DecoderRNN, self).__init__()
        self.hidden_size = hidden_size

        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):
        output = self.embedding(input).view(1, 1, -1)
        output = F.relu(output)
        output, hidden = self.gru(output, hidden)
        output = self.softmax(self.out(output[0]))
        return output, hidden

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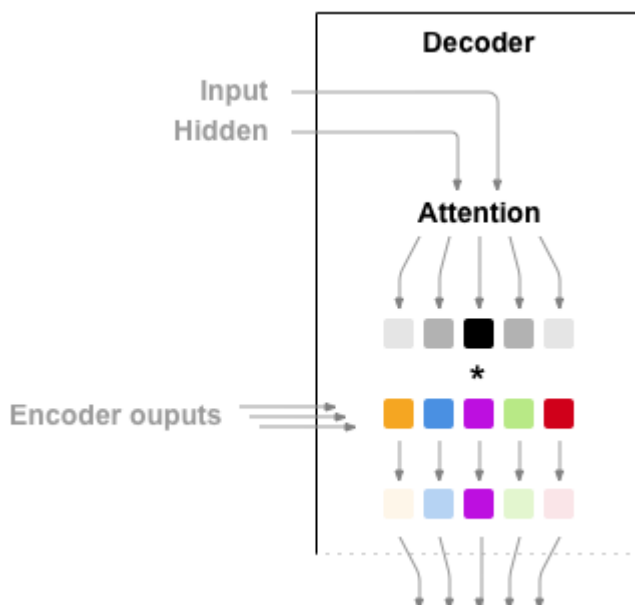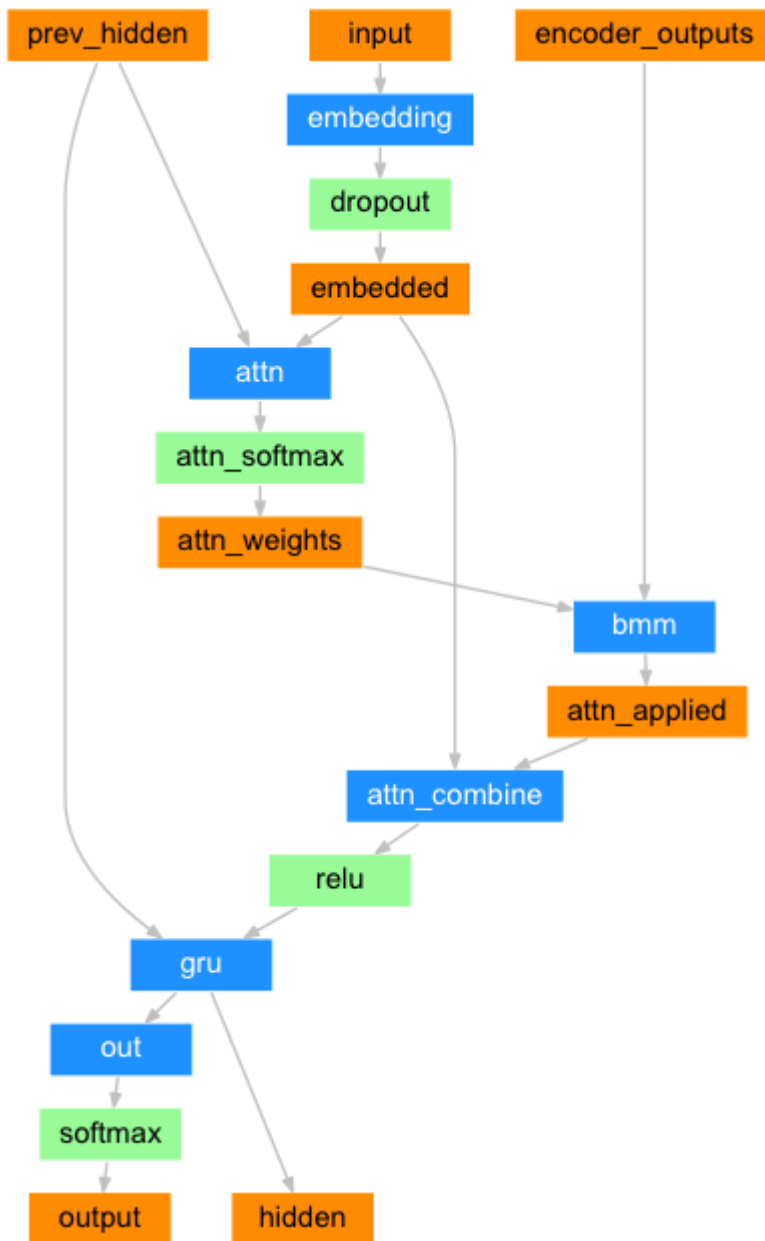
Attention Decoder

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.

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.

```
prev_hidden        input        encoder_outputs
                     │                │
                     ▼                │
                 embedding            │
                     │                │
                     ▼                │
                  dropout             │
                     │                │
                     ▼                │
                 embedded             │
                     │                │
                     ▼                │
                   attn               │
                     │                │
                     ▼                │
               attn_softmax           │
                     │                │
                     ▼                │
               attn_weights           │
                     │                │
                     ▼                ▼
                          bmm
                           │
                           ▼
                      attn_applied
                           │
                           ▼
                   attn_combine
                           │
                           ▼
                         relu
                           │
                           ▼
                         gru
                           │
                           ▼
                         out
                           │
                           ▼
                       softmax
                        │       │
                        ▼       ▼
                    output    hidden
```

```python
class AttnDecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size, dropout_p=0.1, max_length=MAX_LEN
        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_size)
        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):
        embedded = self.embedding(input).view(1, 1, -1)
        embedded = self.dropout(embedded)
```

```python
        attn_weights = F.softmax(
            self.attn(torch.cat((embedded[0], hidden[0]), 1)), dim=1)
        attn_applied = torch.bmm(attn_weights.unsqueeze(0),
                                 encoder_outputs.unsqueeze(0))

        output = torch.cat((embedded[0], attn_applied[0]), 1)
        output = self.attn_combine(output).unsqueeze(0)

        output = F.relu(output)
        output, hidden = self.gru(output, hidden)

        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 `__.

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.

```python
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(-1, 1)


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](#).

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.

```
teacher_forcing_ratio = 0.5


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

    encoder_optimizer.zero_grad()
    decoder_optimizer.zero_grad()

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

    encoder_outputs = torch.zeros(max_length, encoder.hidden_size, device=device)

    loss = 0

    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)

    decoder_hidden = encoder_hidden

    use_teacher_forcing = True if random.random() < teacher_forcing_ratio else Fals

    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 input
            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 history as input

                loss += criterion(decoder_output, target_tensor[di])
                if decoder_input.item() == EOS_token:
                    break

        loss.backward()

        encoder_optimizer.step()
        decoder_optimizer.step()

        return loss.item() / target_length


    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.

```
    def trainIters(encoder, decoder, n_iters, print_every=1000, plot_every=100, learni
        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)
```

```
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, criterion)
    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, print_loss_av

    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.

```
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.

```python
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, device=devic

        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)  # SOS

        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:

```python
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.

```
hidden_size = 200
encoder1 = EncoderRNN(input_lang.n_words, hidden_size).to(device)
attn_decoder1 = AttnDecoderRNN(hidden_size, output_lang.n_words, dropout_p=0.1).to

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

```
1m 9s (- 16m 6s) (5000 6%) 3.4635
2m 15s (- 14m 41s) (10000 13%) 2.8478
3m 22s (- 13m 30s) (15000 20%) 2.5342
4m 29s (- 12m 22s) (20000 26%) 2.3152
5m 37s (- 11m 15s) (25000 33%) 2.1065
6m 44s (- 10m 6s) (30000 40%) 1.9241
7m 51s (- 8m 58s) (35000 46%) 1.7957
8m 58s (- 7m 50s) (40000 53%) 1.6356
10m 5s (- 6m 43s) (45000 60%) 1.5461
11m 13s (- 5m 36s) (50000 66%) 1.4592
12m 20s (- 4m 29s) (55000 73%) 1.3663
13m 28s (- 3m 22s) (60000 80%) 1.3188
14m 35s (- 2m 14s) (65000 86%) 1.2512
15m 42s (- 1m 7s) (70000 93%) 1.1586
16m 49s (- 0m 0s) (75000 100%) 1.1476
```

```
evaluateRandomly(encoder1, attn_decoder1)
```

```
> i m in the mood to talk now .
= je suis maintenant dispose a discuter .
< je suis maintenant la de l . . <EOS>

> you re very good .
= vous etes fort bonnes .
< vous etes fort bonne . <EOS>

> they re spies .
= ce sont des espionnes .
< ils sont medecins . <EOS>

> i am going to play soccer tomorrow .
= demain je jouerai au football .
```

```
        < je vais de me demain . <EOS>

        > i m not your servant .
        = je ne suis pas ton serviteur .
        < je ne suis pas ton merci . <EOS>

        > she is looking for her car keys .
        = elle cherche ses cles de voiture .
        < elle cherche les cles de sa voiture . <EOS>

        > you re very talented .
        = tu es plein de talent .
        < vous etes fort talentueuse . <EOS>

        > you re very intelligent .
        = tu es tres intelligente .
        < vous es tres intelligente . <EOS>

        > i m happy because you re here .
        = je suis heureux parce que vous etes ici .
        < je me rejouis que vous sois ici ici . <EOS>

        > we re smart .
        = nous sommes intelligents .
        < nous sommes intelligents . <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:

```
%matplotlib inline
output_words, attentions = evaluate(
    encoder1, attn_decoder1, "i am sorry .")
plt.matshow(attentions.numpy())
```

```
<matplotlib.image.AxesImage at 0x7f23974e0850>
```

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

```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("i am not ok .")

evaluateAndShowAttention("he is a bad person .")

evaluateAndShowAttention("she is fast .")

evaluateAndShowAttention("we are going tomorrow .")
```
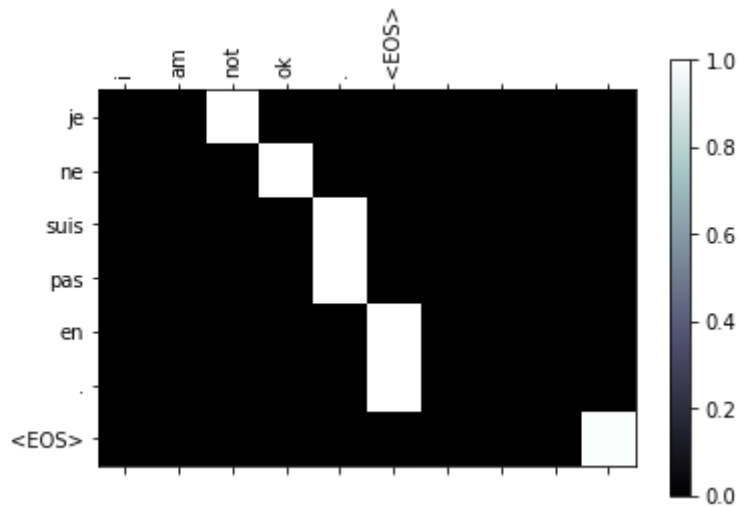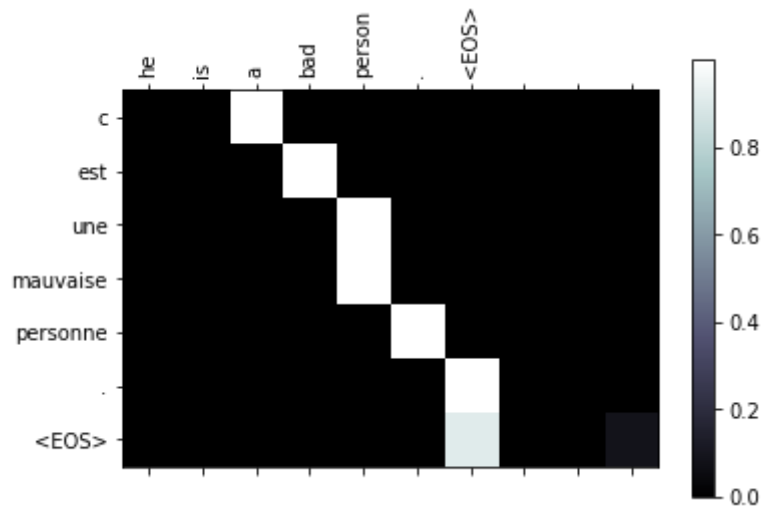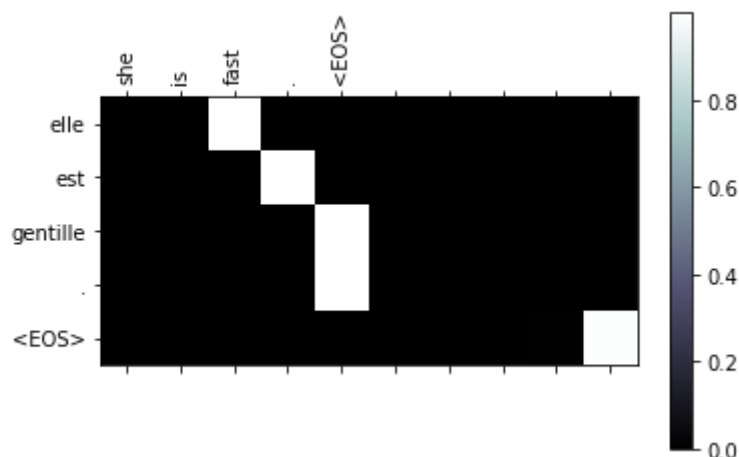
```
input = i am not ok .
output = je ne suis pas en . <EOS>
```



```
input = he is a bad person .
output = c est une mauvaise personne . <EOS>
```



```
input = she is fast .
output = elle est gentille . <EOS>
```
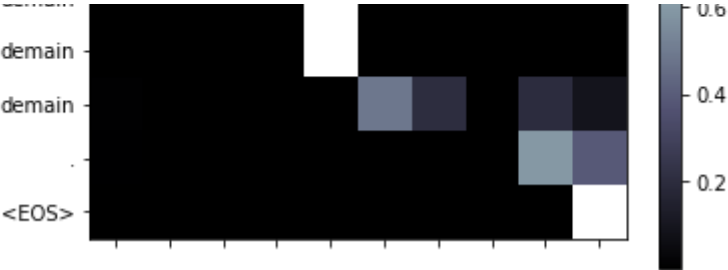


```
input = we are going tomorrow .
output = nous allons demain demain demain . <EOS>
```

✓ 0s   completed at 5:43 PM   ● ✕