Practical PyTorch

Practical PyTorch: Translation with a Sequence to Sequence Network and Attention

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 .
```

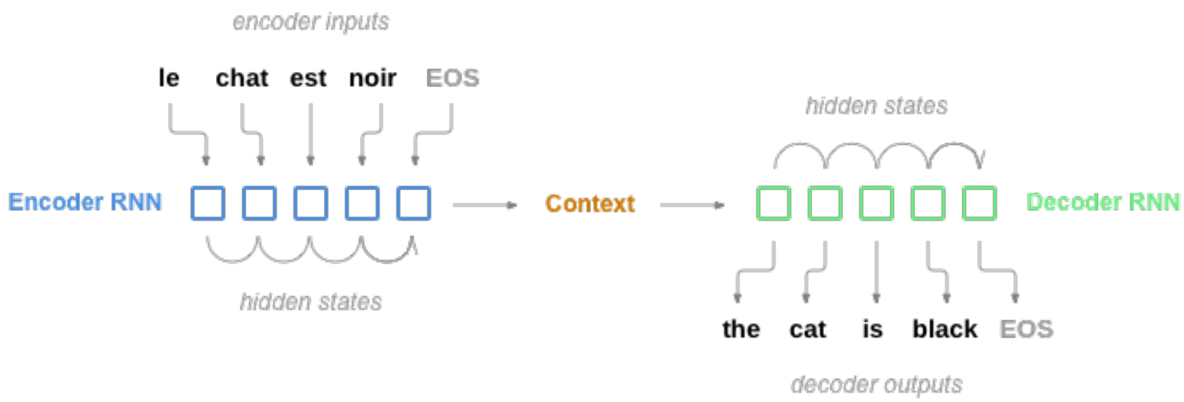
... to varying degrees of success.

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 single vector, and a decoder network unfolds that vector into a new sequence.

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.

Sequence to Sequence Learning

A Sequence to Sequence network, or seq2seq network, or Encoder Decoder network, is a model consisting of two separate RNNs called the **encoder** and **decoder**. The encoder reads an input sequence one item at a time, and outputs a vector at each step. The final output of the encoder is kept as the **context** vector. The decoder uses this context vector to produce a sequence of outputs one step at a time.



When using a single RNN, there is a one-to-one relationship between inputs and outputs. We would quickly run into problems with different sequence orders and lengths that are common during translation. Consider the simple sentence "Je ne suis pas le chat noir" → "I am not the black cat". Many of the words have a pretty direct translation, like "chat" → "cat". However the differing grammars cause words to be in different orders, e.g. "chat noir" and "black cat". There is also the "ne ... pas" → "not" construction that makes the two sentences have different lengths.

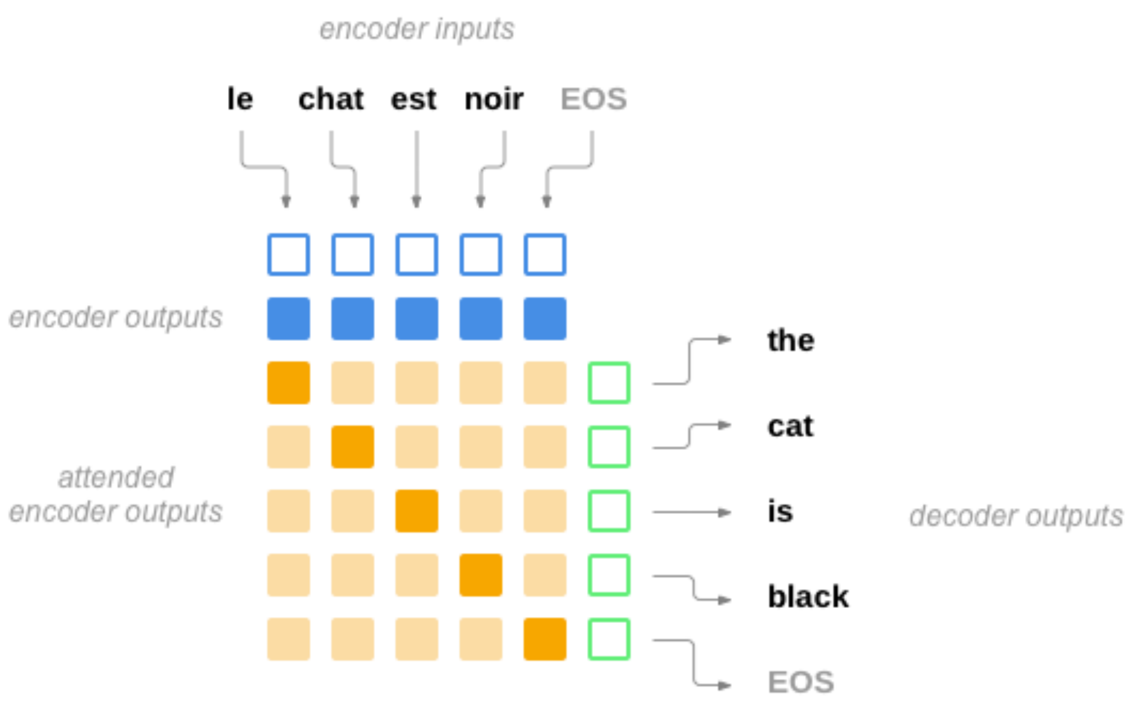
With the seq2seq model, by encoding many inputs into one vector, and decoding from one vector into many outputs, we are freed from the constraints of sequence order and length. The encoded sequence is represented by a single vector, a single point in some N dimensional space of sequences. In an ideal case, this point can be considered the "meaning" of the sequence.

This idea can be extended beyond sequences. Image captioning tasks take an image as input, and output a description of the image (img2seq). Some image generation tasks take a description as input and output a generated image (seq2img). These models can be referred to more generally as "encoder decoder" networks.

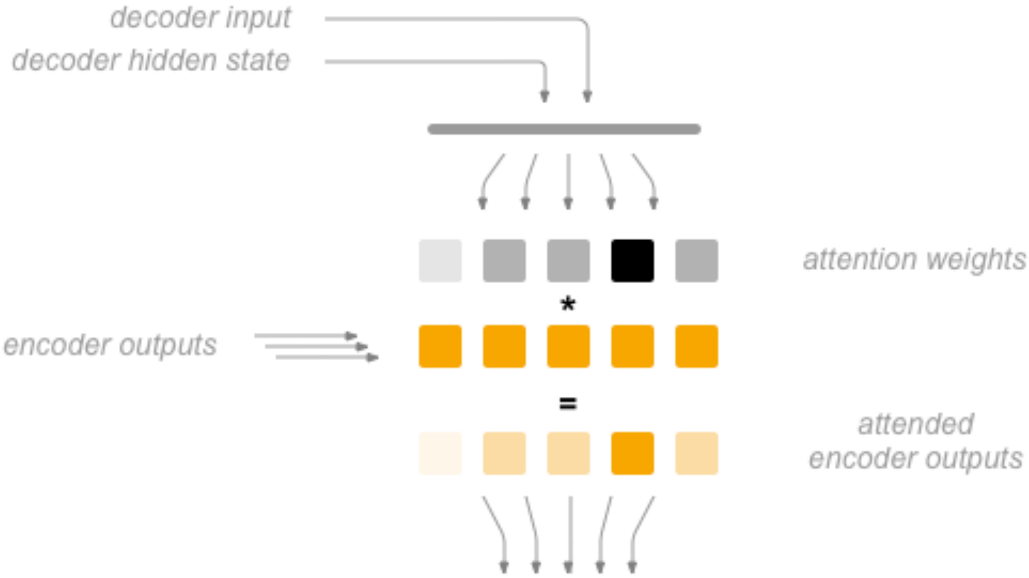
The Attention Mechanism

The fixed-length vector carries the burden of encoding the the entire "meaning" of the input sequence, no matter how long that may be. With all the variance in language, this is a very hard problem. Imagine two nearly identical sentences, twenty words long, with only one word different. Both the encoders and decoders must be nuanced enough to represent that change as a very slightly different point in space.

The **attention mechanism** introduced by Bahdanau et al. addresses this by giving the decoder a way to "pay attention" to parts of the input, rather than relying on a single vector. For every step the decoder can select a different part of the input sentence to consider.



Attention is calculated using the current hidden state and each encoder output, resulting in a vector the same size as the input sequence, called the *attention weights*. These weights are multiplied by the encoder outputs to create a weighted sum of encoder outputs, which is called the *context* vector. The context vector and hidden state are used to predict the next output element.



Requirements

You will need [PyTorch](http://pytorch.org/) to build and train the models, and [matplotlib](https://matplotlib.org/) for plotting training and visualizing attention outputs later. The rest are builtin Python libraries.

```
import unicodedata
import string
import re
import random
import time
import datetime
import math
import socket
hostname = socket.gethostname()

import torch
import torch.nn as nn
from torch.autograd import Variable
from torch import optim
import torch.nn.functional as F
from torch.nn.utils.rnn import pad_packed_sequence, pack_padded_sequence#, masked_cross_entropy
from masked_cross_entropy import *

import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
import numpy as np
%matplotlib inline
```

Here we will also define a constant to decide whether to use the GPU (with CUDA specifically) or the CPU. **If you don't have a GPU, set this to `False`**. Later when we create tensors, this variable will be used to decide whether we keep them on CPU or move them to GPU.

```
USE_CUDA = True
```

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 to the open translation site http://tatoeba.org/ which has downloads available at http://tatoeba.org/eng/downloads - and better yet, someone did the extra work of splitting language pairs into individual text files here: http://www.manythings.org/anki/

The English to French pairs are too big to include in the repo, so download `fra-eng.zip`, extract the text file in there, and rename it to `data/eng-fra.txt` before continuing (for some reason the zipfile is named backwards). The file is a tab separated list of translation pairs:

```
I am cold.    Je suis froid.
```

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.

Indexing words

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`). This class includes a function `trim(min_count)` to remove rare words once they are all counted.

```
PAD_token = 0
SOS_token = 1
EOS_token = 2

class Lang:
    def __init__(self, name):
        self.name = name
        self.trimmed = False
        self.word2index = {}
        self.word2count = {}
        self.index2word = {0: "PAD", 1: "SOS", 2: "EOS"}
        self.n_words = 3 # Count default tokens

    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:
```

```python
            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

    # Remove words below a certain count threshold
    def trim(self, min_count):
        if self.trimmed: return
        self.trimmed = True

        keep_words = []

        for k, v in self.word2count.items():
            if v >= min_count:
                keep_words.append(k)

        print('keep_words %s / %s = %.4f' % (
            len(keep_words), len(self.word2index), len(keep_words) / len(self.word2index)
        ))

        # Reinitialize dictionaries
        self.word2index = {}
        self.word2count = {}
        self.index2word = {0: "PAD", 1: "SOS", 2: "EOS"}
        self.n_words = 3 # Count default tokens

        for word in keep_words:
            self.index_word(word)
```

Reading and decoding files

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 http://stackoverflow.com/a/518232/2809427
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)
    s = re.sub(r"\s+", r" ", s).strip()
    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.

```python
def read_langs(lang1, lang2, reverse=False):
    print("Reading lines...")

    # Read the file and split into lines
#     filename = '../data/%s-%s.txt' % (lang1, lang2)
    filename = '../%s-%s.txt' % (lang1, lang2)
    lines = open(filename).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
```

```python
MIN_LENGTH = 3
MAX_LENGTH = 25

def filter_pairs(pairs):
    filtered_pairs = []
    for pair in pairs:
        if len(pair[0]) >= MIN_LENGTH and len(pair[0]) <= MAX_LENGTH \
            and len(pair[1]) >= MIN_LENGTH and len(pair[1]) <= MAX_LENGTH:
                filtered_pairs.append(pair)
    return filtered_pairs
```

The full process for preparing the data is:

- Read text file and split into lines
- Split lines into pairs and normalize
- Filter to pairs of a certain length
- Make word lists from sentences in pairs

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

    pairs = filter_pairs(pairs)
    print("Filtered to %d pairs" % len(pairs))

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

    print('Indexed %d words in input language, %d words in output' % (input_lang.n_words, output_lang.n_words))
    return input_lang, output_lang, pairs

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

```
Reading lines...
Read 135646 sentence pairs
Filtered to 25706 pairs
Indexing words...
Indexed 6999 words in input language, 4343 words in output
```

Filtering vocabularies

To get something that trains in under an hour, we'll trim the data set a bit. First we will use the `trim` function on each language (defined above) to only include words that are repeated a certain amount of times through the dataset (this softens the difficulty of learning a correct translation for words that don't appear often).

```python
MIN_COUNT = 5

input_lang.trim(MIN_COUNT)
output_lang.trim(MIN_COUNT)
```

```
keep_words 1717 / 6996 = 0.2454
keep_words 1529 / 4340 = 0.3523
```

Filtering pairs

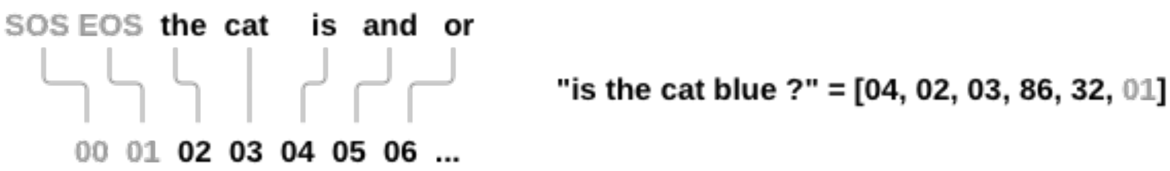
Now we will go back to the set of all sentence pairs and remove those with unknown words.

```
keep_pairs = []

for pair in pairs:
    input_sentence = pair[0]
    output_sentence = pair[1]
    keep_input = True
    keep_output = True

    for word in input_sentence.split(' '):
        if word not in input_lang.word2index:
            keep_input = False
            break

    for word in output_sentence.split(' '):
        if word not in output_lang.word2index:
            keep_output = False
            break

    # Remove if pair doesn't match input and output conditions
    if keep_input and keep_output:
        keep_pairs.append(pair)

print("Trimmed from %d pairs to %d, %.4f of total" % (len(pairs), len(keep_pairs), len(keep_pairs) / len(pairs)))
pairs = keep_pairs
```

```
Trimmed from 25706 pairs to 15896, 0.6184 of total
```

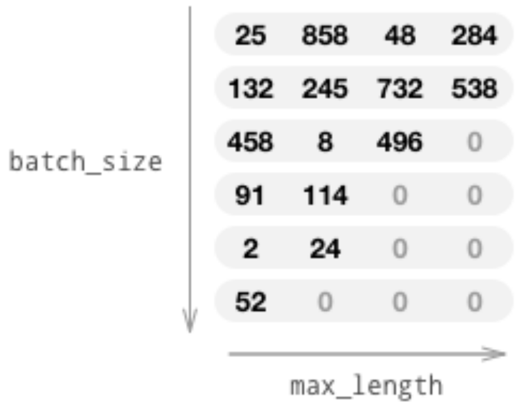
Turning training data into Tensors

To train we need to turn the sentences into something the neural network can understand, which of course means numbers. Each sentence will be split into words and turned into a `LongTensor` which represents the index (from the Lang indexes made earlier) of each word. While creating these tensors we will also append the EOS token to signal that the sentence is over.




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

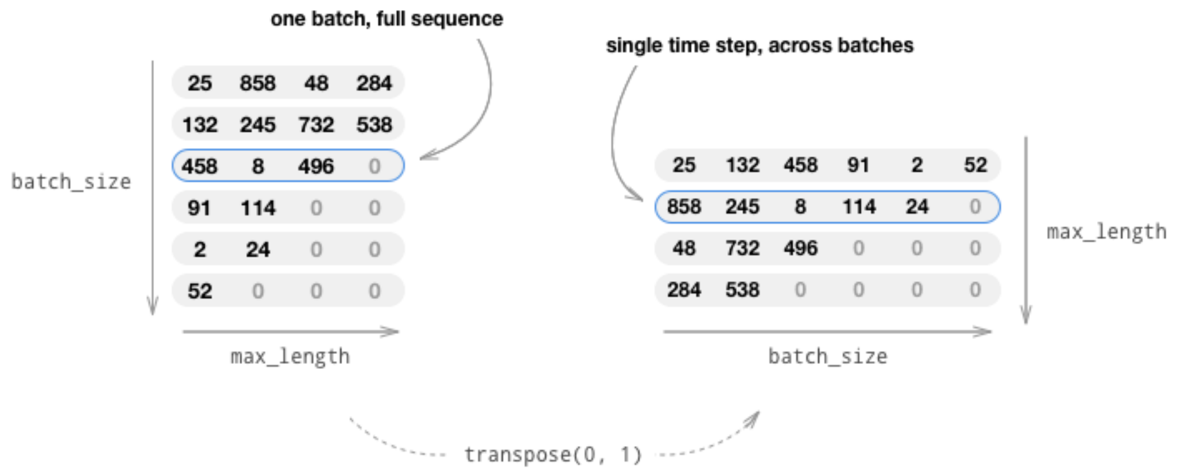
We can make better use of the GPU by training on batches of many sequences at once, but doing so brings up the question of how to deal with sequences of varying lengths. The simple solution is to "pad" the shorter sentences with some padding symbol (in this case `0`), and ignore these padded spots when calculating the loss.




```
# Pad a with the PAD symbol
def pad_seq(seq, max_length):
    seq += [PAD_token for i in range(max_length - len(seq))]
    return seq
```

To create a Variable for a full batch of inputs (and targets) we get a random sample of sequences and pad them all to the length of the longest sequence. We'll keep track of the lengths of each batch in order to un-pad later.

Initializing a `LongTensor` with an array (batches) of arrays (sequences) gives us a `(batch_size x max_len)` tensor - selecting the first dimension gives you a single batch, which is a full sequence. When training the model we'll want a single time step at once, so we'll transpose to `(max_len x batch_size)`. Now selecting along the first dimension returns a single time step across batches.



```
def random_batch(batch_size):
    input_seqs = []
    target_seqs = []

    # Choose random pairs
    for i in range(batch_size):
        pair = random.choice(pairs)
        input_seqs.append(indexes_from_sentence(input_lang, pair[0]))
        target_seqs.append(indexes_from_sentence(output_lang, pair[1]))

    # Zip into pairs, sort by length (descending), unzip
    seq_pairs = sorted(zip(input_seqs, target_seqs), key=lambda p: len(p[0]), reverse=True)
    input_seqs, target_seqs = zip(*seq_pairs)

    # For input and target sequences, get array of lengths and pad with 0s to max length
    input_lengths = [len(s) for s in input_seqs]
    input_padded = [pad_seq(s, max(input_lengths)) for s in input_seqs]
    target_lengths = [len(s) for s in target_seqs]
    target_padded = [pad_seq(s, max(target_lengths)) for s in target_seqs]

    # Turn padded arrays into (batch_size x max_len) tensors, transpose into (max_len x batch_size)
    input_var = Variable(torch.LongTensor(input_padded)).transpose(0, 1)
    target_var = Variable(torch.LongTensor(target_padded)).transpose(0, 1)

    if USE_CUDA:
        input_var = input_var.cuda()
        target_var = target_var.cuda()

    return input_var, input_lengths, target_var, target_lengths
```

We can test this to see that it will return a `(max_len x batch_size)` tensor for input and target sentences, along with a corresponding list of batch lenghts for each (which we will use for masking later).

```
random_batch(2)
```

```
(Variable containing:
   88    92
   44   208
  107   297
  634    14
   14     2
    2     0
 [torch.cuda.LongTensor of size 6x2 (GPU 0)], [6, 5], Variable
containing:
    50     50
  1128     19
   436     26
   969      4
     4      2
     2      0
 [torch.cuda.LongTensor of size 6x2 (GPU 0)], [6, 5])
```

Building the models

The Encoder

The encoder will take a batch of word sequences, a `LongTensor` of size `(max_len x batch_size)`, and output an encoding for each word, a `FloatTensor` of size `(max_len x batch_size x hidden_size)`.

The word inputs are fed through an embedding layer `nn.Embedding` to create an embedding for each word, with size `seq_len x hidden_size` (as if it was a batch of words). This is resized to `seq_len x 1 x hidden_size` to fit the expected input of the GRU layer `nn.GRU`. The GRU will return both an output sequence of size `seq_len x hidden_size`.

```python
class EncoderRNN(nn.Module):
    def __init__(self, input_size, hidden_size, n_layers=1, dropout=0.1):
        super(EncoderRNN, self).__init__()

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

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

    def forward(self, input_seqs, input_lengths, hidden=None):
        # Note: we run this all at once (over multiple batches of multiple sequences)
        embedded = self.embedding(input_seqs)
        packed = torch.nn.utils.rnn.pack_padded_sequence(embedded, input_lengths)
        outputs, hidden = self.gru(packed, hidden)
        outputs, output_lengths = torch.nn.utils.rnn.pad_packed_sequence(outputs) # unpack (back to padded)
        outputs = outputs[:, :, :self.hidden_size] + outputs[:, : ,self.hidden_size:] # Sum bidirectional outputs
        return outputs, hidden
```

Attention Decoder

Interpreting the Bahdanau et al. model

[Neural Machine Translation by Jointly Learning to Align and Translate](#) (Dzmitry Bahdanau, Kyunghyun Cho, Yoshua Bengio) introduced the idea of using attention for seq2seq translation.

Each decoder output is conditioned on the previous outputs and some $\mathbf{x}$, where $\mathbf{x}$ consists of the current hidden state (which takes into account previous outputs) and the attention "context", which is calculated below. The function g is a fully-connected layer with a nonlinear activation, which takes as input the values y_{i-1}, s_i, and c_i concatenated.

$$p(y_i \mid \{y_1, \ldots, y_{i-1}\}, \mathbf{x}) = g(y_{i-1}, s_i, c_i) \tag{1}$$

The current hidden state s_i is calculated by an RNN f with the last hidden state s_{i-1}, last decoder output value y_{i-1}, and context vector c_i.

In the code, the RNN will be a `nn.GRU` layer, the hidden state s_i will be called `hidden`, the output y_i called `output`, and context c_i called `context`.

$$s_i = f(s_{i-1}, y_{i-1}, c_i) \tag{2}$$

The context vector c_i is a weighted sum of all encoder outputs, where each weight a_{ij} is the amount of "attention" paid to the corresponding encoder output h_j.

$$c_i = \sum_{j=1}^{T_x} a_{ij} h_j \tag{3}$$

... where each weight a_{ij} is a normalized (over all steps) attention "energy" e_{ij} ...

$$a_{ij} = \frac{exp(e_{ij})}{\sum_{k=1}^{T} exp(e_{ik})} \tag{4}$$

... where each attention energy is calculated with some function a (such as another linear layer) using the last hidden state s_{i-1} and that particular encoder output h_j:

$$e_{ij} = a(s_{i-1}, h_j) \tag{5}$$

Interpreting the Luong et al. models

Effective Approaches to Attention-based Neural Machine Translation (Minh-Thang Luong, Hieu Pham, Christopher D. Manning) describe a few more attention models that offer improvements and simplifications. They describe a few "global attention" models, the distinction between them being the way the attention scores are calculated.

The general form of the attention calculation relies on the target (decoder) side hidden state and corresponding source (encoder) side state, normalized over all states to get values summing to 1:

$$a_t(s) = align(h_t, \bar{h}_s) = \frac{exp(score(h_t, \bar{h}_s))}{\sum_{s'} exp(score(h_t, \bar{h}_{s'}))} \tag{6}$$

The specific "score" function that compares two states is either *dot*, a simple dot product between the states; *general*, a a dot product between the decoder hidden state and a linear transform of the encoder state; or *concat*, a dot product between a new parameter v_a and a linear transform of the states concatenated together.

$$score(h_t, \bar{h}_s) = \begin{cases} h_t^\top \bar{h}_s & dot \\ h_t^\top \mathbf{W}_a \bar{h}_s & general \\ v_a^\top \mathbf{W}_a [h_t; \bar{h}_s] & concat \end{cases} \tag{7}$$

The modular definition of these scoring functions gives us an opportunity to build specific attention module that can switch between the different score methods. The input to this module is always the hidden state (of the decoder RNN) and set of encoder outputs.

Implementing an attention module

```
class Attn(nn.Module):
    def __init__(self, method, hidden_size):
        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.v = nn.Parameter(torch.FloatTensor(1, hidden_size))

    def forward(self, hidden, encoder_outputs):
        max_len = encoder_outputs.size(0)
        this_batch_size = encoder_outputs.size(1)

        # Create variable to store attention energies
        attn_energies = Variable(torch.zeros(this_batch_size, max_len)) # B x S

        if USE_CUDA:
            attn_energies = attn_energies.cuda()
```

```

        # For each batch of encoder outputs
        for b in range(this_batch_size):
            # Calculate energy for each encoder output
            for i in range(max_len):
                attn_energies[b, i] = self.score(hidden[:, b], encoder_outputs[i, b].unsqueeze(0))

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

    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)
            energy = hidden.dot(energy)
            return energy

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

Implementing the Bahdanau et al. model

In summary our decoder should consist of four main parts - an embedding layer turning an input word into a vector; a layer to calculate the attention energy per encoder output; a RNN layer; and an output layer.

The decoder's inputs are the last RNN hidden state s_{i-1}, last output y_{i-1}, and all encoder outputs h_*.

- embedding layer with inputs y_{i-1}
  - `embedded = embedding(last_rnn_output)`
- attention layer a with inputs (s_{i-1}, h_j) and outputs e_{ij}, normalized to create a_{ij}

    - `attn_energies[j] = attn_layer(last_hidden, encoder_outputs[j])`
    - `attn_weights = normalize(attn_energies)`
- context vector c_i as an attention-weighted average of encoder outputs
    - `context = sum(attn_weights * encoder_outputs)`
- RNN layer(s) f with inputs (s_{i-1}, y_{i-1}, c_i) and internal hidden state, outputting s_i
    - `rnn_input = concat(embedded, context)`
    - `rnn_output, rnn_hidden = rnn(rnn_input, last_hidden)`
- an output layer g with inputs (y_{i-1}, s_i, c_i), outputting y_i
    - `output = out(embedded, rnn_output, context)`

```
class BahdanauAttnDecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size, n_layers=1, dropout_p=0.1):
        super(BahdanauAttnDecoderRNN, 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 = Attn('concat', hidden_size)
        self.gru = nn.GRU(hidden_size, 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: we run this one step at a time
        # TODO: FIX BATCHING

```

```
        # 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
        context = context.transpose(0, 1) # 1 x B 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
```

Now we can build a decoder that plugs this Attn module in after the RNN to calculate attention weights, and apply those weights to the encoder outputs to get a context vector.

```
class LuongAttnDecoderRNN(nn.Module):
    def __init__(self, attn_model, hidden_size, output_size, n_layers=1, dropout=0.1):
        super(LuongAttnDecoderRNN, self).__init__()

        # Keep for reference
        self.attn_model = attn_model
        self.hidden_size = hidden_size
        self.output_size = output_size
```

```
        self.n_layers = n_layers
        self.dropout = dropout

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

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

    def forward(self, input_seq, last_hidden, encoder_outputs):
        # Note: we run this one step at a time

        # Get the embedding of the current input word (last output word)
        batch_size = input_seq.size(0)
        embedded = self.embedding(input_seq)
        embedded = self.embedding_dropout(embedded)
        embedded = embedded.view(1, batch_size, self.hidden_size) # S=1 x B x N

        # Get current hidden state from input word and last hidden state
        rnn_output, hidden = self.gru(embedded, last_hidden)

        # Calculate attention from current RNN state and all encoder outputs;
        # apply to encoder outputs to get weighted average
        attn_weights = self.attn(rnn_output, encoder_outputs)
        context = attn_weights.bmm(encoder_outputs.transpose(0, 1)) # B x S=1 x N

```

```
        # Attentional vector using the RNN hidden state and context vector
        # concatenated together (Luong eq. 5)
        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
        concat_input = torch.cat((rnn_output, context), 1)
        concat_output = F.tanh(self.concat(concat_input))

        # Finally predict next token (Luong eq. 6, without softmax)
        output = self.out(concat_output)

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

Testing the models

To make sure the encoder and decoder modules are working (and working together) we'll do a full test with a small batch.

```
small_batch_size = 3
input_batches, input_lengths, target_batches, target_lengths = random_batch(small_batch_size)

print('input_batches', input_batches.size()) # (max_len x batch_size)
print('target_batches', target_batches.size()) # (max_len x batch_size)
```

```
input_batches torch.Size([7, 3])
target_batches torch.Size([8, 3])
```

Create models with a small size (a good idea for eyeball inspection):

```
small_hidden_size = 8
small_n_layers = 2

encoder_test = EncoderRNN(input_lang.n_words, small_hidden_size, small_n_layers)
decoder_test = LuongAttnDecoderRNN('general', small_hidden_size, output_lang.n_words, small_n_layers)

if USE_CUDA:
    encoder_test.cuda()
    decoder_test.cuda()
```

To test the encoder, run the input batch through to get per-batch encoder outputs:

```
encoder_outputs, encoder_hidden = encoder_test(input_batches, input_lengths, None)

print('encoder_outputs', encoder_outputs.size()) # max_len x batch_size x hidden_size
print('encoder_hidden', encoder_hidden.size()) # n_layers * 2 x batch_size x hidden_size
```

```
encoder_outputs torch.Size([7, 3, 8])
encoder_hidden torch.Size([4, 3, 8])
```

Then starting with a SOS token, run word tokens through the decoder to get each next word token. Instead of doing this with the whole sequence, it is done one at a time, to support using it's own predictions to make the next prediction. This will be one time step at a time, but batched per time step. In order to get this to work for short padded sequences, the batch size is going to get smaller each time.

```python
max_target_length = max(target_lengths)

# Prepare decoder input and outputs
decoder_input = Variable(torch.LongTensor([SOS_token] * small_batch_size))
decoder_hidden = encoder_hidden[:decoder_test.n_layers] # Use last (forward) hidden state from encoder
all_decoder_outputs = Variable(torch.zeros(max_target_length, small_batch_size, decoder_test.output_size))

if USE_CUDA:
    all_decoder_outputs = all_decoder_outputs.cuda()
    decoder_input = decoder_input.cuda()

# Run through decoder one time step at a time
for t in range(max_target_length):
    decoder_output, decoder_hidden, decoder_attn = decoder_test(
        decoder_input, decoder_hidden, encoder_outputs
    )
    all_decoder_outputs[t] = decoder_output # Store this step's outputs
    decoder_input = target_batches[t] # Next input is current target

# Test masked cross entropy loss
loss = masked_cross_entropy(
    all_decoder_outputs.transpose(0, 1).contiguous(),
    target_batches.transpose(0, 1).contiguous(),
    target_lengths
)
print('loss', loss.data[0])
```

```
loss 7.343282222747803
```

Training

Defining a training iteration

To train we first run the input sentence through the encoder word by word, and keep track of every output and the latest hidden state. Next the decoder is given the last hidden state of the decoder as its first hidden state, and the `<sos>` token as its first input. From there we iterate to predict a next token from the decoder.

Teacher Forcing vs. Scheduled Sampling

"Teacher Forcing", or maximum likelihood sampling, means using the real target outputs as each next input when training. The alternative is using the decoder's own guess as the next input. Using teacher forcing may cause the network 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 - you could think of it as having learned how to listen to the teacher's instructions, without learning how to venture out on its own.

The solution to the teacher-forcing "problem" is known as [Scheduled Sampling](), which simply alternates between using the target values and predicted values when training. We will randomly choose to use teacher forcing with an if statement while training - sometimes we'll feed use real target as the input (ignoring the decoder's output), sometimes we'll use the decoder's output.

```
def train(input_batches, input_lengths, target_batches, target_lengths, encoder, decoder, encoder_optimizer, decoder_optimizer, criterion, max_length=MAX_LENGTH):

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

    # Run words through encoder
    encoder_outputs, encoder_hidden = encoder(input_batches, input_lengths, None)

    # Prepare input and output variables
    decoder_input = Variable(torch.LongTensor([SOS_token] * batch_size))
    decoder_hidden = encoder_hidden[:decoder.n_layers] # Use last (forward) hidden state from encoder
```

```

    max_target_length = max(target_lengths)
    all_decoder_outputs = Variable(torch.zeros(max_target_length, batch_size, decoder.output_size))

    # Move new Variables to CUDA
    if USE_CUDA:
        decoder_input = decoder_input.cuda()
        all_decoder_outputs = all_decoder_outputs.cuda()

    # Run through decoder one time step at a time
    for t in range(max_target_length):
        decoder_output, decoder_hidden, decoder_attn = decoder(
            decoder_input, decoder_hidden, encoder_outputs
        )

        all_decoder_outputs[t] = decoder_output
        decoder_input = target_batches[t] # Next input is current target

    # Loss calculation and backpropagation
    loss = masked_cross_entropy(
        all_decoder_outputs.transpose(0, 1).contiguous(), # -> batch x seq
        target_batches.transpose(0, 1).contiguous(), # -> batch x seq
        target_lengths
    )
    loss.backward()

    # Clip gradient norms
    ec = torch.nn.utils.clip_grad_norm(encoder.parameters(), clip)
    dc = torch.nn.utils.clip_grad_norm(decoder.parameters(), clip)

    # Update parameters with optimizers
    encoder_optimizer.step()
    decoder_optimizer.step()

```

```
    return loss.data[0], ec, dc
```

Running training

With everything in place we can actually initialize a network and start training.

To start, we initialize models, optimizers, a loss function (criterion), and set up variables for plotting and tracking progress:

```
# Configure models
attn_model = 'dot'
hidden_size = 500
n_layers = 2
dropout = 0.1
batch_size = 100
batch_size = 50

# Configure training/optimization
clip = 50.0
teacher_forcing_ratio = 0.5
learning_rate = 0.0001
decoder_learning_ratio = 5.0
n_epochs = 50000
epoch = 0
plot_every = 20
print_every = 100
evaluate_every = 1000

# Initialize models
encoder = EncoderRNN(input_lang.n_words, hidden_size, n_layers, dropout=dropout)
decoder = LuongAttnDecoderRNN(attn_model, hidden_size, output_lang.n_words, n_layers, dropout=dropout)

# Initialize optimizers and criterion
encoder_optimizer = optim.Adam(encoder.parameters(), lr=learning_rate)
decoder_optimizer = optim.Adam(decoder.parameters(), lr=learning_rate * decoder_learning_ratio)
criterion = nn.CrossEntropyLoss()

# Move models to GPU
```

```
if USE_CUDA:
    encoder.cuda()
    decoder.cuda()

import sconce
job = sconce.Job('seq2seq-translate', {
    'attn_model': attn_model,
    'n_layers': n_layers,
    'dropout': dropout,
    'hidden_size': hidden_size,
    'learning_rate': learning_rate,
    'clip': clip,
    'teacher_forcing_ratio': teacher_forcing_ratio,
    'decoder_learning_ratio': decoder_learning_ratio,
})
job.plot_every = plot_every
job.log_every = print_every

# 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
```

```
Starting job 59739ec4f8e1c2083c28a9f6 at 2017-07-22 20:11:42
```

Plus helper functions to print time elapsed and estimated time remaining, given the current time and progress.

```
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))
```

Evaluating the network

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

```
def evaluate(input_seq, max_length=MAX_LENGTH):
    input_lengths = [len(input_seq)]
    input_seqs = [indexes_from_sentence(input_lang, input_seq)]
    input_batches = Variable(torch.LongTensor(input_seqs), volatile=True).transpose(0, 1)

    if USE_CUDA:
        input_batches = input_batches.cuda()

    # Set to not-training mode to disable dropout
    encoder.train(False)
    decoder.train(False)

    # Run through encoder
    encoder_outputs, encoder_hidden = encoder(input_batches, input_lengths, None)

    # Create starting vectors for decoder
    decoder_input = Variable(torch.LongTensor([SOS_token]), volatile=True) # SOS
    decoder_hidden = encoder_hidden[:decoder.n_layers] # Use last (forward) hidden state from encoder

    if USE_CUDA:
        decoder_input = decoder_input.cuda()

    # Store output words and attention states
    decoded_words = []
    decoder_attentions = torch.zeros(max_length + 1, max_length + 1)

    # Run through decoder
    for di in range(max_length):
```

```
        decoder_output, decoder_hidden, decoder_attention = decoder(
            decoder_input, decoder_hidden, encoder_outputs
        )
        decoder_attentions[di,:decoder_attention.size(2)] += decoder_attention.squeeze(0).squeeze(0).cpu().data

        # Choose top word from output
        topv, topi = decoder_output.data.topk(1)
        ni = topi[0][0]
        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()

    # Set back to training mode
    encoder.train(True)
    decoder.train(True)

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

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

```
def evaluate_randomly():
    [input_sentence, target_sentence] = random.choice(pairs)
    evaluate_and_show_attention(input_sentence, target_sentence)
```

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:

```
import io
import torchvision
from PIL import Image
import visdom
vis = visdom.Visdom()

def show_plot_visdom():
    buf = io.BytesIO()
    plt.savefig(buf)
    buf.seek(0)
    attn_win = 'attention (%s)' % hostname
    vis.image(torchvision.transforms.ToTensor()(Image.open(buf)), win=attn_win, opts={'title': attn_win})
```

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

```
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))

    show_plot_visdom()
    plt.show()
    plt.close()
```

```
def evaluate_and_show_attention(input_sentence, target_sentence=None):
    output_words, attentions = evaluate(input_sentence)
    output_sentence = ' '.join(output_words)
    print('>', input_sentence)
    if target_sentence is not None:
        print('=', target_sentence)
    print('<', output_sentence)

    show_attention(input_sentence, output_words, attentions)

    # Show input, target, output text in visdom
    win = 'evaluted (%s)' % hostname
    text = '<p>&gt; %s</p><p>= %s</p><p>&lt; %s</p>' % (input_sentence, target_sentence, output_sentence)
    vis.text(text, win=win, opts={'title': win})
```

Putting it all together

TODO Run `train_epochs` for `n_epochs`

To actually train, we call the train function many times, printing a summary as we go.

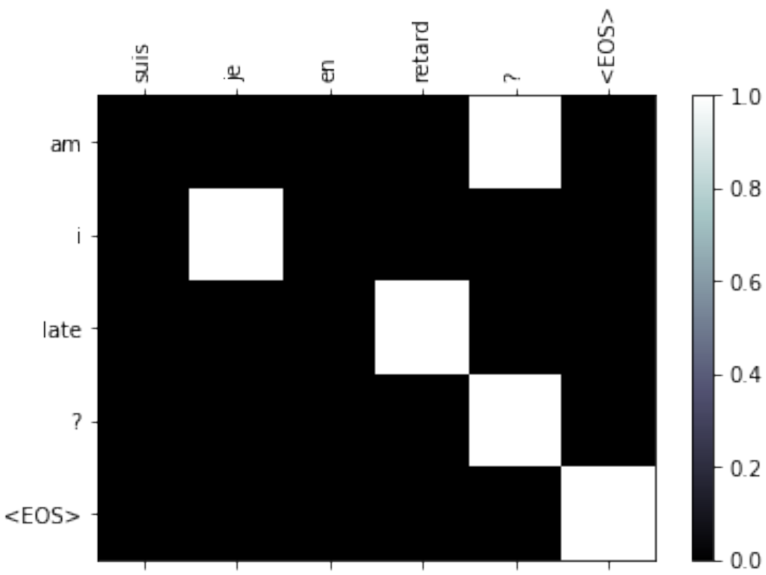
Note: If you're running this notebook you can **train, interrupt, evaluate, and come back to continue training**. Simply run the notebook starting from the following cell (running from the previous cell will reset the models).

```
# Begin!
ecs = []
dcs = []
eca = 0
dca = 0

while epoch < n_epochs:
    epoch += 1

    # Get training data for this cycle
    input_batches, input_lengths, target_batches, target_lengths = random_batch(batch_size)

```

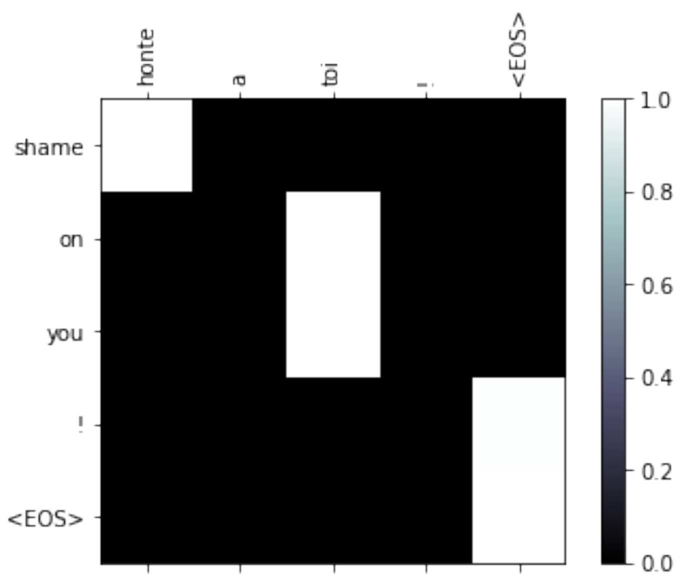
```
    # Run the train function
    loss, ec, dc = train(
        input_batches, input_lengths, target_batches, target_lengths,
        encoder, decoder,
        encoder_optimizer, decoder_optimizer, criterion
    )

    # Keep track of loss
    print_loss_total += loss
    plot_loss_total += loss
    eca += ec
    dca += dc

    job.record(epoch, loss)

    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), epoch, epoch / n_epochs * 100, print_loss_avg)
        print(print_summary)

    if epoch % evaluate_every == 0:
        evaluate_randomly()

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

        # TODO: Running average helper
        ecs.append(eca / plot_every)
        dcs.append(dca / plot_every)
        ecs_win = 'encoder grad (%s)' % hostname
        dcs_win = 'decoder grad (%s)' % hostname
        vis.line(np.array(ecs), win=ecs_win, opts={'title': ecs_win})
        vis.line(np.array(dcs), win=dcs_win, opts={'title': dcs_win})
        eca = 0
```

```
            dca = 0
```

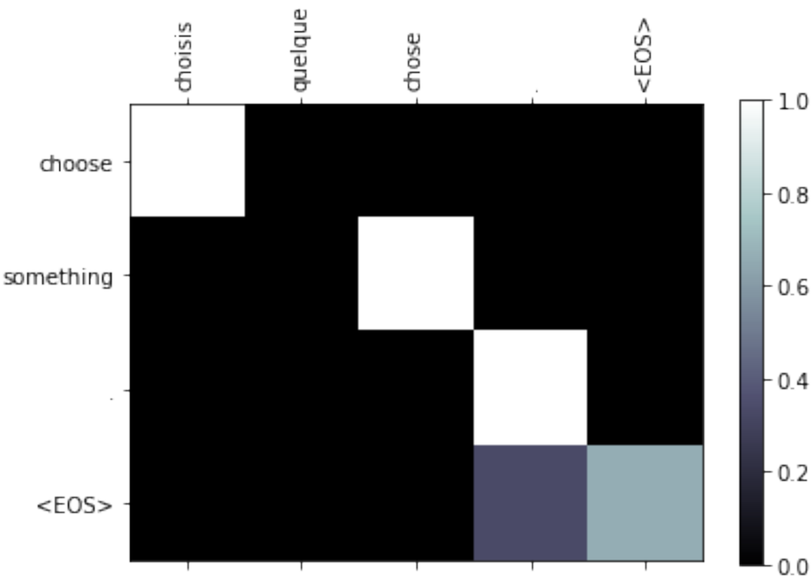
```
[log] 1m 50s (100) 3.1331
1m 50s (- 921m 56s) (100 0%) 3.8196
[log] 3m 41s (200) 2.3766
3m 41s (- 921m 4s) (200 0%) 2.7289
[log] 5m 35s (300) 2.1629
5m 35s (- 926m 34s) (300 0%) 2.2523
[log] 7m 28s (400) 1.9996
7m 28s (- 926m 21s) (400 0%) 1.9320
[log] 9m 20s (500) 1.5955
9m 20s (- 924m 47s) (500 1%) 1.6854
[log] 11m 13s (600) 1.2429
11m 13s (- 924m 11s) (600 1%) 1.4429
[log] 13m 5s (700) 1.2304
13m 5s (- 922m 26s) (700 1%) 1.2527
[log] 14m 57s (800) 0.9507
14m 57s (- 919m 49s) (800 1%) 1.1110
[log] 16m 49s (900) 0.8307
16m 49s (- 917m 34s) (900 1%) 0.9817
[log] 18m 39s (1000) 0.7994
18m 39s (- 914m 34s) (1000 2%) 0.8726
> suis je en retard ?
= am i late ?
< am i late ? <EOS>
```



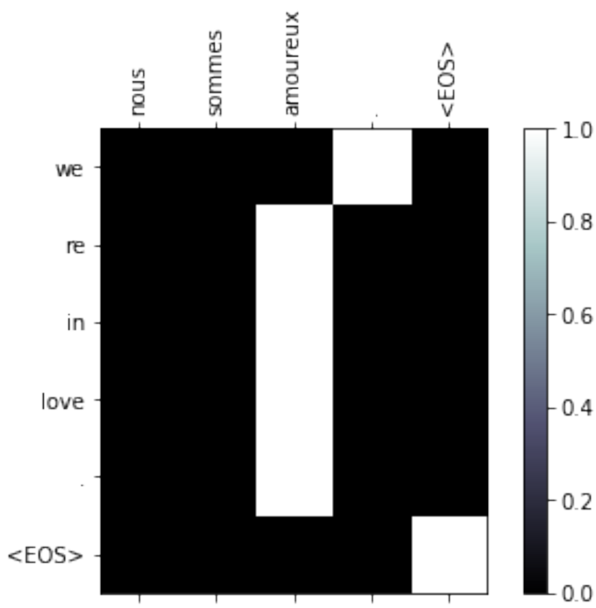
```
[log] 20m 29s (1100) 0.6578
20m 29s (- 911m 11s) (1100 2%) 0.7791
[log] 22m 19s (1200) 0.6510
22m 19s (- 907m 38s) (1200 2%) 0.6962
[log] 24m 10s (1300) 0.5559
24m 10s (- 905m 40s) (1300 2%) 0.6159
[log] 26m 0s (1400) 0.4897
26m 0s (- 903m 1s) (1400 2%) 0.5736
[log] 27m 50s (1500) 0.5131
27m 50s (- 900m 5s) (1500 3%) 0.5190
[log] 29m 38s (1600) 0.3948
29m 38s (- 896m 52s) (1600 3%) 0.4632
[log] 31m 27s (1700) 0.6653
31m 27s (- 893m 44s) (1700 3%) 0.4410
[log] 33m 15s (1800) 0.3286
33m 15s (- 890m 39s) (1800 3%) 0.3999
[log] 35m 5s (1900) 0.4149
35m 5s (- 888m 17s) (1900 3%) 0.3684
[log] 36m 54s (2000) 0.2788
36m 54s (- 885m 52s) (2000 4%) 0.3466
> honte a toi !
= shame on you .
< shame on you ! <EOS>
```



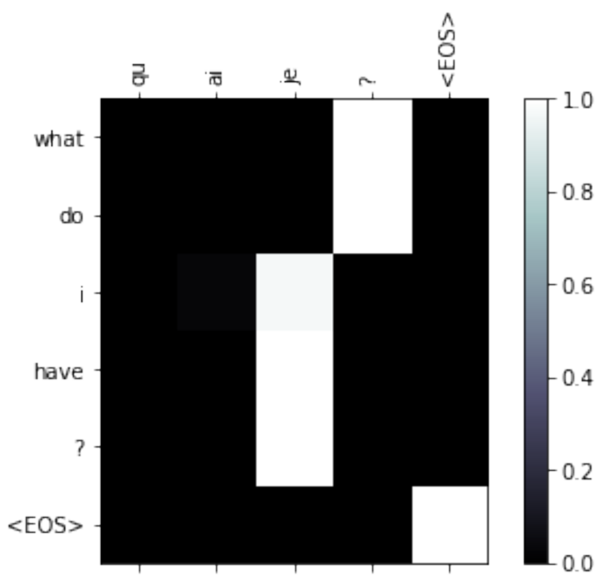
```
[log] 38m 44s (2100) 0.3325
38m 44s (- 883m 46s) (2100 4%) 0.3377
[log] 40m 36s (2200) 0.2391
40m 36s (- 882m 12s) (2200 4%) 0.3217
[log] 42m 25s (2300) 0.2144
42m 25s (- 879m 49s) (2300 4%) 0.3013
[log] 44m 14s (2400) 0.2987
44m 15s (- 877m 38s) (2400 4%) 0.2743
[log] 46m 4s (2500) 0.2795
46m 4s (- 875m 27s) (2500 5%) 0.2610
[log] 47m 53s (2600) 0.2676
47m 53s (- 873m 0s) (2600 5%) 0.2380
[log] 49m 43s (2700) 0.1816
49m 43s (- 871m 10s) (2700 5%) 0.2199
[log] 51m 35s (2800) 0.2438
51m 35s (- 869m 43s) (2800 5%) 0.2171
[log] 53m 25s (2900) 0.2003
53m 25s (- 867m 44s) (2900 5%) 0.1989
[log] 55m 16s (3000) 0.2235
55m 16s (- 865m 59s) (3000 6%) 0.1900
> choisis quelque chose .
= choose something .
< choose something . <EOS>
```



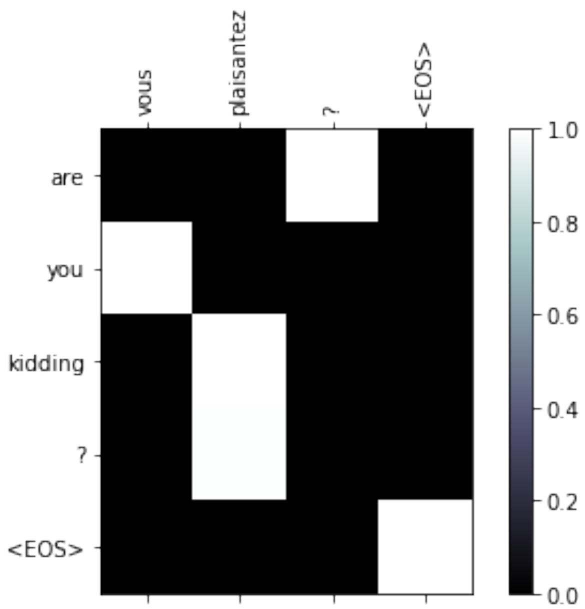
```
[log] 57m 8s (3100) 0.2420
57m 8s (- 864m 34s) (3100 6%) 0.1877
[log] 58m 57s (3200) 0.1424
58m 57s (- 862m 22s) (3200 6%) 0.1783
[log] 60m 50s (3300) 0.1371
60m 50s (- 860m 59s) (3300 6%) 0.1750
[log] 62m 41s (3400) 0.1539
62m 41s (- 859m 21s) (3400 6%) 0.1679
[log] 64m 30s (3500) 0.1167
64m 30s (- 857m 8s) (3500 7%) 0.1695
[log] 66m 23s (3600) 0.1849
66m 23s (- 855m 44s) (3600 7%) 0.1630
[log] 68m 16s (3700) 0.1372
68m 16s (- 854m 25s) (3700 7%) 0.1544
[log] 70m 8s (3800) 0.1163
70m 8s (- 852m 44s) (3800 7%) 0.1434
[log] 72m 0s (3900) 0.1499
72m 0s (- 851m 7s) (3900 7%) 0.1415
[log] 73m 51s (4000) 0.1129
73m 51s (- 849m 18s) (4000 8%) 0.1405
> nous sommes amoureux .
= we re in love .
< we re in love . <EOS>
```



```
[log] 75m 43s (4100) 0.1106
75m 43s (- 847m 48s) (4100 8%) 0.1315
[log] 77m 34s (4200) 0.0593
77m 34s (- 846m 0s) (4200 8%) 0.1353
[log] 79m 27s (4300) 0.1601
79m 27s (- 844m 29s) (4300 8%) 0.1256
[log] 81m 17s (4400) 0.1076
81m 17s (- 842m 29s) (4400 8%) 0.1285
[log] 83m 8s (4500) 0.1967
83m 8s (- 840m 42s) (4500 9%) 0.1237
[log] 84m 59s (4600) 0.1156
84m 59s (- 838m 49s) (4600 9%) 0.1175
[log] 86m 51s (4700) 0.0809
86m 51s (- 837m 13s) (4700 9%) 0.1118
[log] 88m 41s (4800) 0.0821
88m 41s (- 835m 13s) (4800 9%) 0.1115
[log] 90m 32s (4900) 0.1044
90m 32s (- 833m 18s) (4900 9%) 0.1140
[log] 92m 23s (5000) 0.0773
92m 23s (- 831m 35s) (5000 10%) 0.1076
> qu ai je ?
= what do i have ?
< what do i have ? <EOS>
```



```
[log] 94m 14s (5100) 0.0806
94m 14s (- 829m 41s) (5100 10%) 0.1055
[log] 96m 6s (5200) 0.0678
96m 6s (- 827m 57s) (5200 10%) 0.1025
[log] 97m 59s (5300) 0.1065
97m 59s (- 826m 30s) (5300 10%) 0.1026
[log] 99m 49s (5400) 0.1059
99m 49s (- 824m 29s) (5400 10%) 0.1007
[log] 101m 40s (5500) 0.1084
101m 40s (- 822m 41s) (5500 11%) 0.0991
[log] 103m 32s (5600) 0.1498
103m 32s (- 820m 54s) (5600 11%) 0.0985
[log] 105m 23s (5700) 0.0675
105m 23s (- 819m 6s) (5700 11%) 0.1009
[log] 107m 15s (5800) 0.1340
107m 15s (- 817m 21s) (5800 11%) 0.0985
[log] 109m 7s (5900) 0.0902
109m 7s (- 815m 39s) (5900 11%) 0.0970
[log] 110m 59s (6000) 0.0942
110m 59s (- 813m 57s) (6000 12%) 0.0997
> vous plaisantez ?
= are you kidding ?
< are you kidding ? <EOS>
```



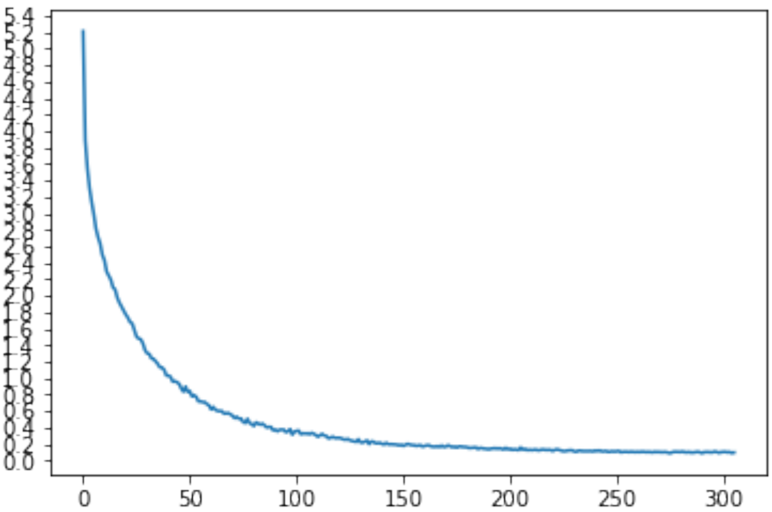

```
[log] 112m 52s (6100) 0.0712
112m 52s (- 812m 17s) (6100 12%) 0.0981
```

Plotting training loss

Plotting is done with matplotlib, using the array `plot_losses` that was created while training.

```
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)
```

```
<matplotlib.figure.Figure at 0x7fc5852c0a20>
```




```
output_words, attentions = evaluate("je suis trop froid .")
plt.matshow(attentions.numpy())
show_plot_visdom()
```

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

```
evaluate_and_show_attention("elle est trop petit .")
```

```
evaluate_and_show_attention("je ne crains pas de mourir .")
```

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

```
evaluate_and_show_attention("est le chien vert aujourd hui ?")
```

```
evaluate_and_show_attention("le chat me parle .")
```

```
evaluate_and_show_attention("des centaines de personnes furent arretees ici .")
```

```
evaluate_and_show_attention("des centaines de chiens furent arretees ici .")
```

```
evaluate_and_show_attention("ce fromage est prepare a partir de lait de chevre .")
```

Exercises

- Try with a different dataset
  - Another language pair
  - Human → Machine (e.g. IOT commands)
  - Chat → Response
  - Question → Answer
- Replace the embedding 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