NLP FROM SCRATCH: TRANSLATION WITH A SEQ ENCE TO SEQ ENCE NETWORK AND ATTENTION

Author: Sean Ro ertson

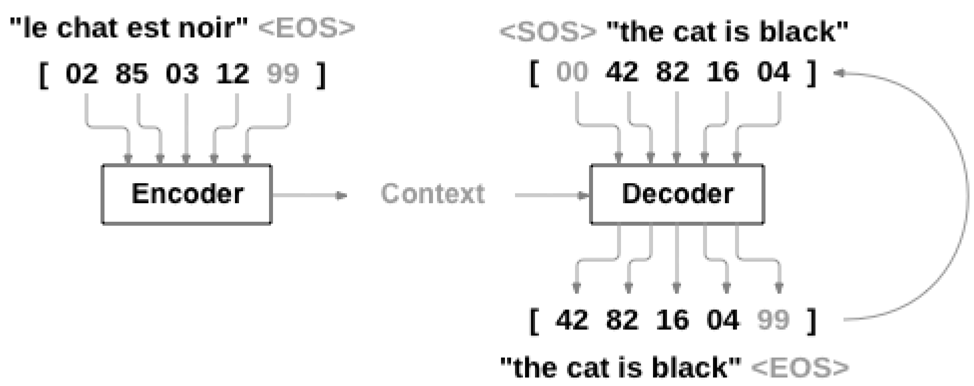
This is the thir an final tutorial on oing "NLP From Scratch", where we write our own classes an functions to preprocess the ata to o our NLP mo eling tasks. We hope after you complete this tutorial that you'll procee to learn how *torchtext* can han le much of this preprocessing for you in the three tutorials imme iately following this one.

In this pro ect we will e 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 egrees of success.

This is ma e possi le y the simple ut powerful i ea of the sequence to sequence network, in which two recurrent neural networks work together to transform one sequence to another. An enco er network con enses an input sequence into a vector, an a eco er network unfol s that vector into a new sequence.



To improve upon this mo el we'll use an attention mechanism, which lets the eco er learn to focus over a specific range of the input sequence.

Recommen e Rea ing:

I assume you have at least installe PyTorch, know Python, an un erstan Tensors:

- https://pytorch.org/ For installation instructions
- Deep Learning with PyTorch: A 60 Minute Blitz to get starte with PyTorch in general
- Learning PyTorch with Examples for a wi e an   eep overview
- PyTorch for Former Torch sers if you are former Lua Torch user

It woul also e useful to know a out Sequence to Sequence networks an how they work:

- Learning Phrase Representations using RNN Enco er-Deco er for Statistical Machine Translation
- Sequence to Sequence Learning with Neural Networks
- Neural Machine Translation y Jointly Learning to Align an Translate
- A Neural Conversational Mo el

You will also fin the previous tutorials on NLP From Scratch: Classifying Names with a Character-Level RNN an NLP From Scratch: Generating Names with a Character-Level RNN helpful as those concepts are very similar to the Enco er an Deco er mo els, respectively.

Re uirements

```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")
```

Loa ing ata files

The ata for this pro ect is a set of many thousan s of English to French translation pairs.

This question on Open Data Stack Exchange pointe me to the open translation site https://tatoe a.org/ which has ownloa s availa le at https://tatoe a.org/eng/ ownloa s - an etter yet, someone i the extra work of splitting language pairs into in ivi ual text files here: https://www.manythings.org/anki/

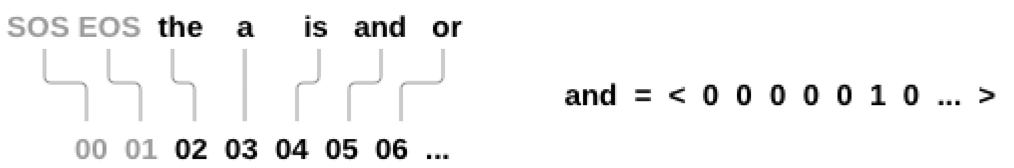
The English to French pairs are too ig to inclu e in the repo, so ownloa to `data/eng-fra.txt` efore continuing. The file is a ta separate list of translation pairs:

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

• NOTE

Downloa the ata from here an extract it to the current irectory.

Similar to the character enco ing use in the character-level RNN tutorials, we will e representing each wor in a language as a one-hot vector, or giant vector of zeros except for a single one (at the in ex of the wor). Compare to the ozens of characters that might exist in a language, there are many many more wor s, so the enco ing vector is much larger. We will however cheat a it an trim the ata to only use a few thousan wor s per language.



We'll nee a unique in ex per wor to use as the inputs an targets of the networks later. To keep track of all this we will use a helper class calle `Lang` which has wor → in ex (`word2index`) an in ex → wor (`index2word`) ictionaries, as well as a count of each wor `word2count` which will e use to replace rare wor s later.

```python
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 nico e, to simplify we will turn nico e characters to ASCII, make everything lowercase, an trim most punctuation.

```python
# 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 rea the ata file we will split the file into lines, an then split lines into pairs. The files are all English → Other Language, so if we want to translate from Other Language → English I a e the `reverse` flag to reverse the pairs.

```python
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 an we want to train something quickly, we'll trim the ata set to only relatively short an simple sentences. Here the maximum length is 10 wor s (that inclu es en ing punctuation) an we're filtering to sentences that translate to the form "I am" or "He is" etc. (accounting for apostrophes replace earlier).

```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 ata is:

- Rea text file an split into lines, split lines into pairs
- Normalize text, filter y length an content
- Make wor 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


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

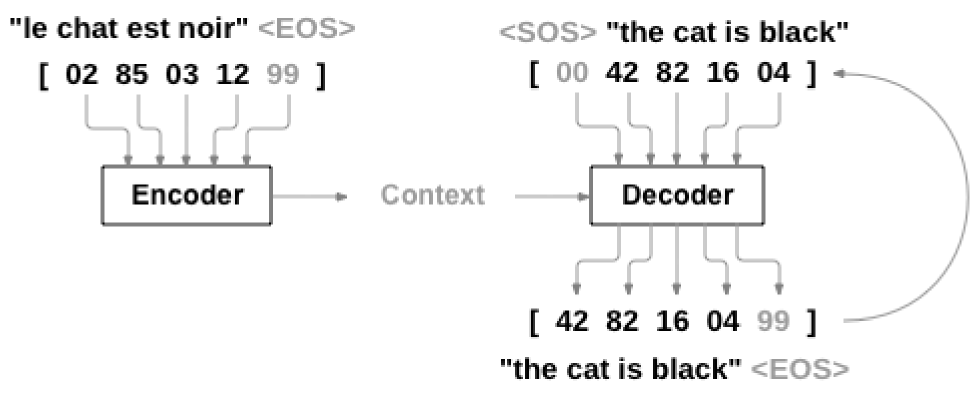
Out:

```
Reading lines...
Read 135842 sentence pairs
Trimmed to 10599 sentence pairs
Counting words...
Counted words:
fra 4345
eng 2803
['il n est vraiment pas heureux .', 'he is far from happy .']
```

The Se 2Se Mo el

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

A Sequence to Sequence network, or seq2seq network, or Enco er Deco er network, is a mo el consisting of two RNNs calle the enco er an eco er. The enco er rea s an input sequence an outputs a single vector, an the eco er rea s that vector to pro uce an output sequence.



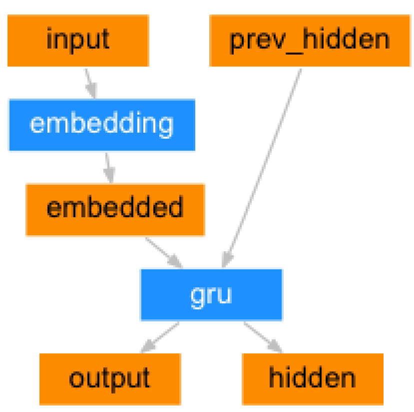
 nlike sequence pre iction with a single RNN, where every input correspon s to an output, the seq2seq mo el frees us from sequence length an or er, which makes it i eal for translation etween two languages.

Consi er the sentence “Je ne suis pas le chat noir” → “I am not the lack cat”. Most of the wor s in the input sentence have a irect translation in the output sentence, ut are in slightly ifferent or ers, e.g. “chat noir” an “ lack cat”. Because of the “ne/pas” construction there is also one more wor in the input sentence. It woul   e ifficult to pro uce a correct translation irectly from the sequence of input wor s.

With a seq2seq mo el the enco er creates a single vector which, in the i eal case, enco es the “meaning” of the input sequence into a single vector — a single point in some N imensional space of sentences.

The Enco er

The enco er of a seq2seq network is a RNN that outputs some value for every wor from the input sentence. For every input wor the enco er outputs a vector an a hi en state, an uses the hi en state for the next input wor .



```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)
```

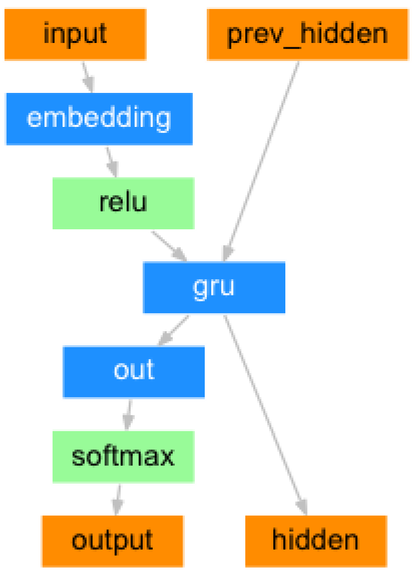
The Deco er

The eco er is another RNN that takes the enco er output vector(s) an outputs a sequence of wor s to create the translation.

Sim le Deco er

In the simplest seq2seq eco er we use only last output of the enco er. This last output is sometimes calle the *context vector* as it enco es context from the entire sequence. This context vector is use as the initial hi en state of the eco er.

At every step of eco ing, the eco er is given an input token an hi en state. The initial input token is the start-of-string `<SOS>` token, an the first hi en state is the context vector (the enco er's last hi en state).



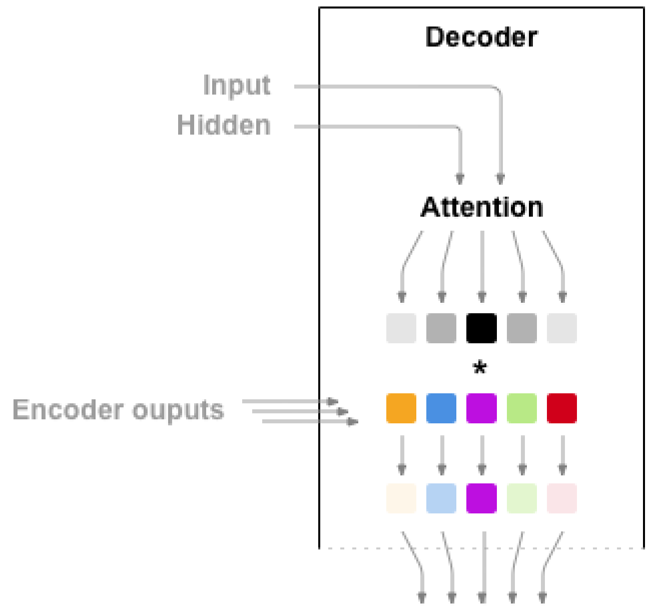

```python
class DecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size):
        super(DecoderRNN, self).__init__()
        self.hidden_size = hidden_size

        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)
```

I encourage you to train an o serve the results of this mo el, ut to save space we'll e going straight for the gol an intro ucing the Attention Mechanism.

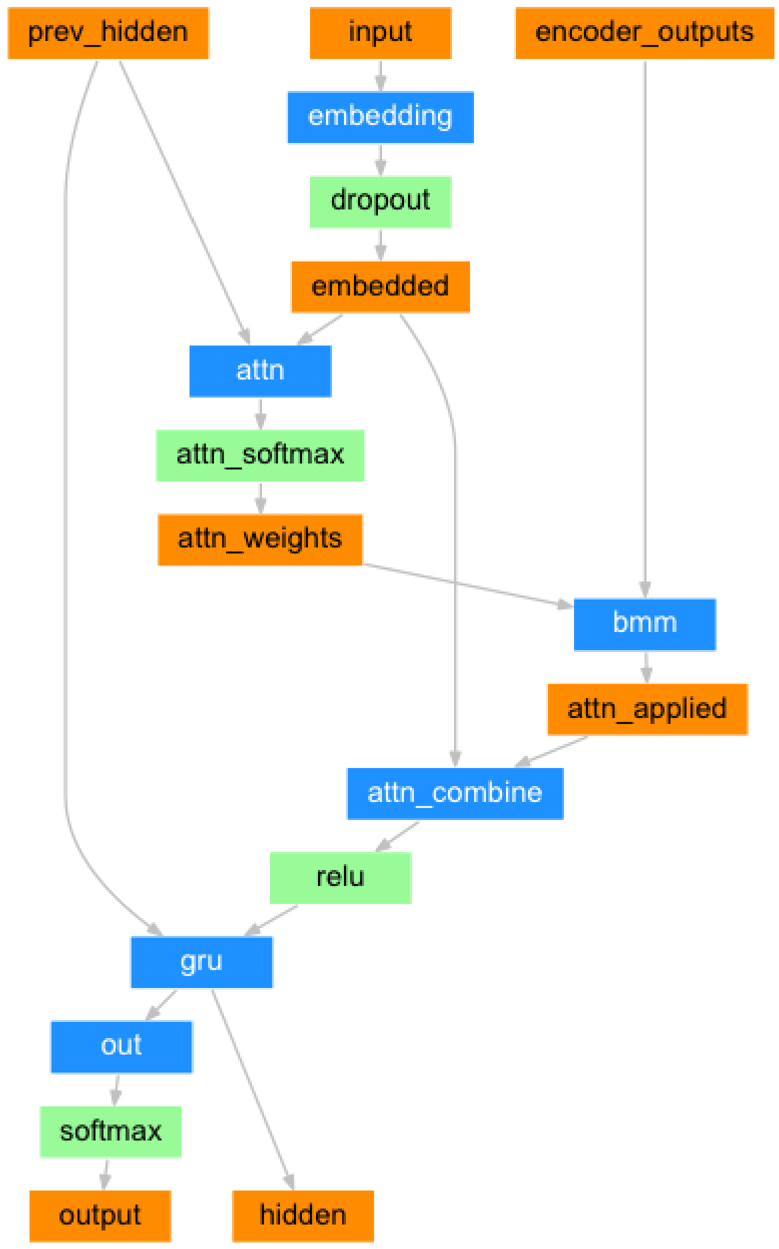
Attention Deco er

If only the context vector is passe etween the enco er an eco er, that single vector carries the ur en of enco ing the entire sentence.

Attention allows the eco er network to "focus" on a ifferent part of the enco er's outputs for every step of the eco er's own outputs. First we calculate a set of *attention weights*. These will e multiplie y the enco er output vectors to create a weighte com ination. The result (calle `attn_applied` in the co e) shoul contain information a out that specific part of the input sequence, an thus help the eco er choose the right output wor s.



Calculating the attention weights is one with another fee -forwar layer `attn`, using the eco er's input an hi en state as inputs. Because there are sentences of all sizes in the training ata, to actually create an train this layer we have to choose a maximum sentence length (input length, for enco er 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.



```python
class AttnDecoderRNN(nn.Module):
    def __init__(self, hidden_size, output_size, dropout_p=0.1, max_length=MAX_LENGTH):
        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)

        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 aroun the length limitation y using a relative position approach. Rea a out "local attention" in Effective Approaches to Attention- ase Neural Machine Translation.

Training

Pre aring Training Data

To train, for each pair we will nee an input tensor (in exes of the wor s in the input sentence) an target tensor (in exes of the wor s in the target sentence). While creating these vectors we will appen the EOS token to oth 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 Mo el

To train we run the input sentence through the enco er, an keep track of every output an the latest hi en state. Then the eco er is given the `<SOS>` token as its first input, an the last hi en state of the enco er as its first hi en state.

"Teacher forcing" is the concept of using the real target outputs as each next input, instea of using the eco er's guess as the next input. sing teacher forcing causes it to converge faster ut when the traine network is exploite , it may exhi it insta ility.

You can o serve outputs of teacher-force networks that rea with coherent grammar ut wan er far from the correct translation - intuitively it has learne to represent the output grammar an can "pick up" the meaning once the teacher tells it the first few wor s, ut it has not properly learne how to create the sentence from the translation in the first place.

Because of the free om PyTorch's autogra gives us, we can ran omly 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, decoder_optimizer, criterion,
max_length=MAX_LENGTH):
    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 False

    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
```

This is a helper function to print time elapse an estimate time remaining given the current time an progress %.

```
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 an criterion
- Create set of training pairs
- Start empty losses array for plotting

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

```
def trainIters(encoder, decoder, n_iters, print_every=1000, plot_every=100, learning_rate=0.01):
    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, 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_avg))

        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 one with matplotli , using the array of loss values `plot_losses` save 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, ut there are no targets so we simply fee the eco er's pre ictions ack to itself for each step. Every time it pre icts a wor we a it to the output string, an if it pre icts the EOS token we stop there. We also store the eco er's attention outputs for isplay later.

```
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=device)

        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 ran om sentences from the training set an print out the input, target, an output to make some su ective quality u gements:

```
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 an Evaluating

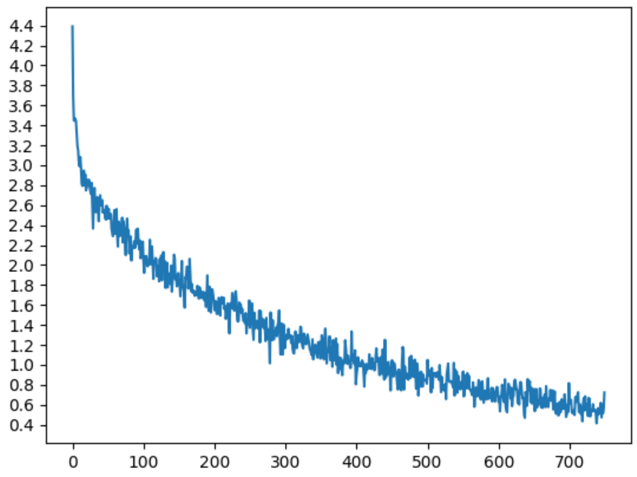
With all these helper functions in place (it looks like extra work, ut it makes it easier to run multiple experiments) we can actually initialize a network an start training.

Remem er that the input sentences were heavily filtere . For this small ataset we can use relatively small networks of 256 hi en no es an a single GR layer. After a out 40 minutes on a MacBook CP we'll get some reasona le results.

• NOTE

If you run this note ook you can train, interrupt the kernel, evaluate, an continue training later. Comment out the lines where the enco er an eco er are initialize an run `trainIters` again.

```
hidden_size = 256
encoder1 = EncoderRNN(input_lang.n_words, hidden_size).to(device)
attn_decoder1 = AttnDecoderRNN(hidden_size, output_lang.n_words, dropout_p=0.1).to(device)

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



Out:

```
1m 41s (- 23m 43s) (5000 6%) 2.8480
3m 6s (- 20m 11s) (10000 13%) 2.3068
4m 31s (- 18m 7s) (15000 20%) 1.9700
5m 57s (- 16m 23s) (20000 26%) 1.7543
7m 22s (- 14m 44s) (25000 33%) 1.5498
8m 45s (- 13m 8s) (30000 40%) 1.3591
10m 10s (- 11m 37s) (35000 46%) 1.2204
11m 35s (- 10m 8s) (40000 53%) 1.0970
12m 59s (- 8m 39s) (45000 60%) 0.9850
14m 23s (- 7m 11s) (50000 66%) 0.9071
15m 49s (- 5m 45s) (55000 73%) 0.8525
17m 14s (- 4m 18s) (60000 80%) 0.7445
18m 39s (- 2m 52s) (65000 86%) 0.6999
20m 7s (- 1m 26s) (70000 93%) 0.6234
21m 32s (- 0m 0s) (75000 100%) 0.5701
```

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

Out:

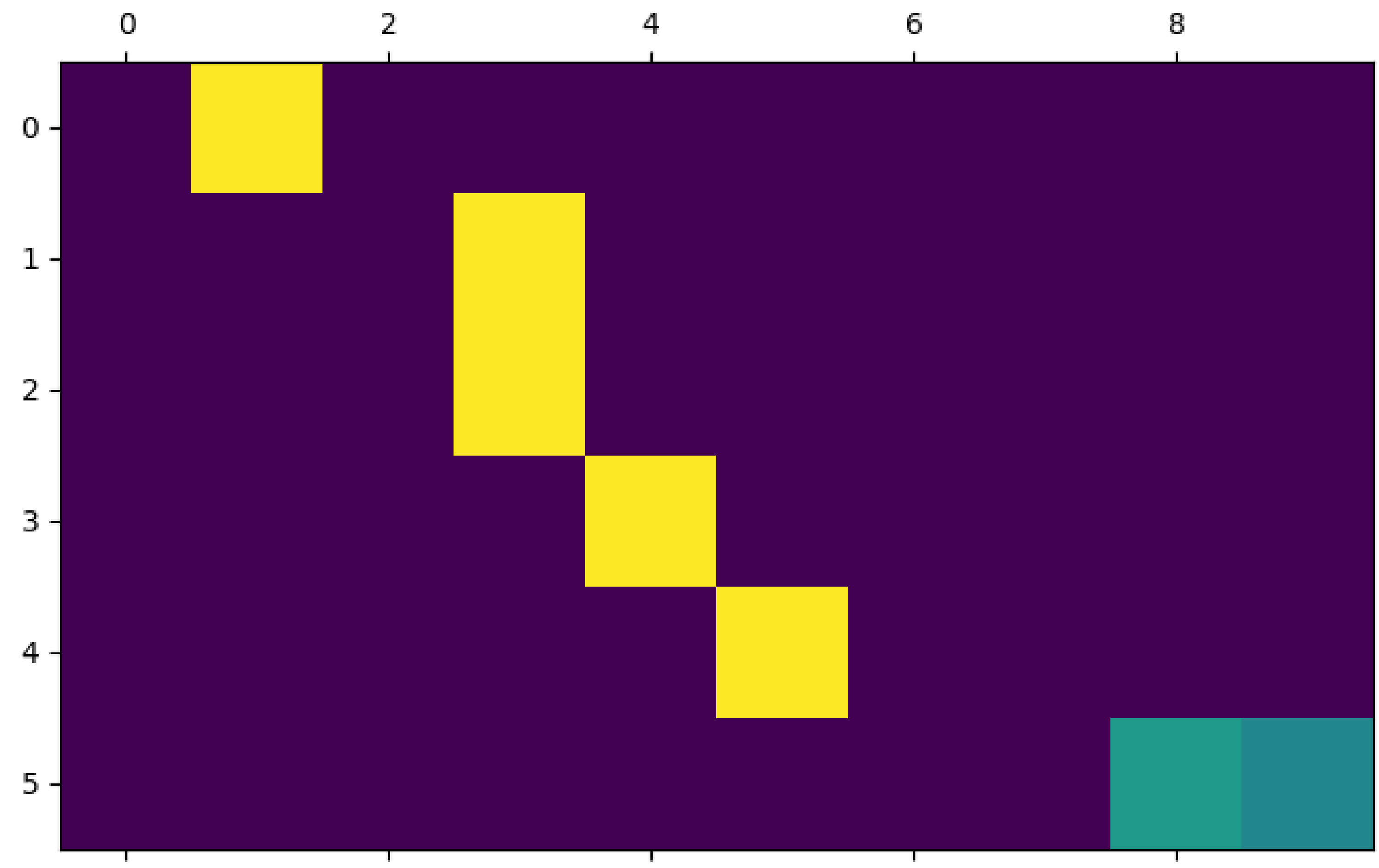
```
> vous etes hors de controle .
= you re out of control .
< you re out of control . <EOS>

> j ai de l interet pour la musique .
= i am interested in music .
< i m interested in music . <EOS>

> je suis impulsif .
= i m impulsive .
< i m impulsive . <EOS>

> ils sont jaloux de nous .
= they are jealous of us .
< they are jealous of us . <EOS>
```

Visualizing Attention

A useful property of the attention mechanism is its highly interpreta le outputs. Because it is use to weight specific enco er outputs of the input sequence, we can imagine looking where the network is focuse most at each time step.

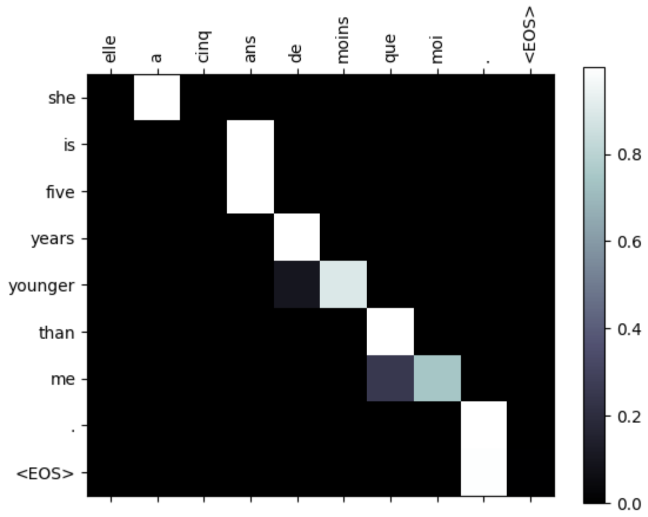
You coul simply run `plt.matshow(attentions)` to see attention output isplaye as a matrix, with the columns eing input steps an rows eing output steps:

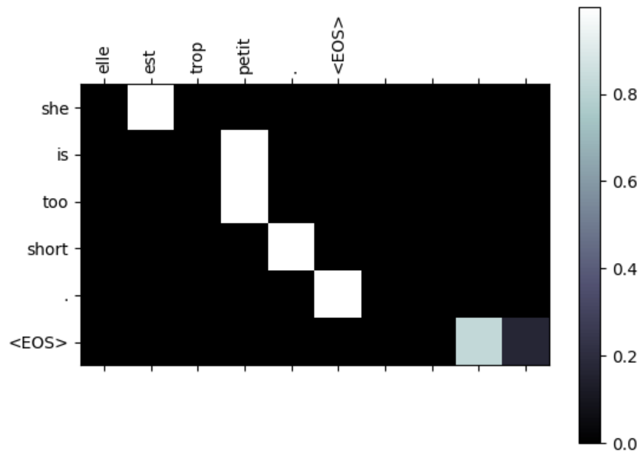
```
output_words, attentions = evaluate(
    encoder1, attn_decoder1, "je suis trop froid .")
plt.matshow(attentions.numpy())
```

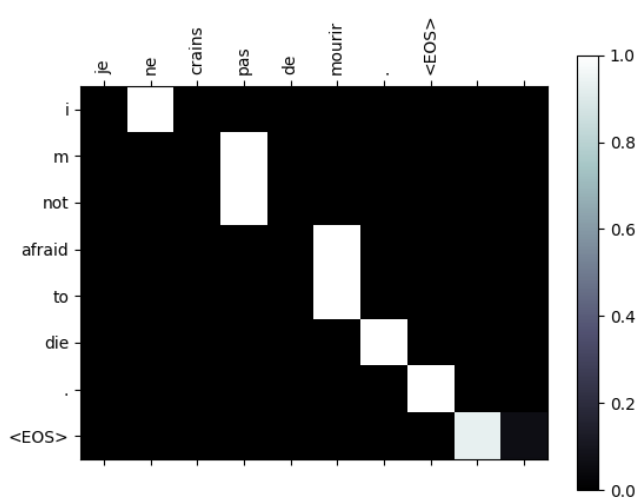


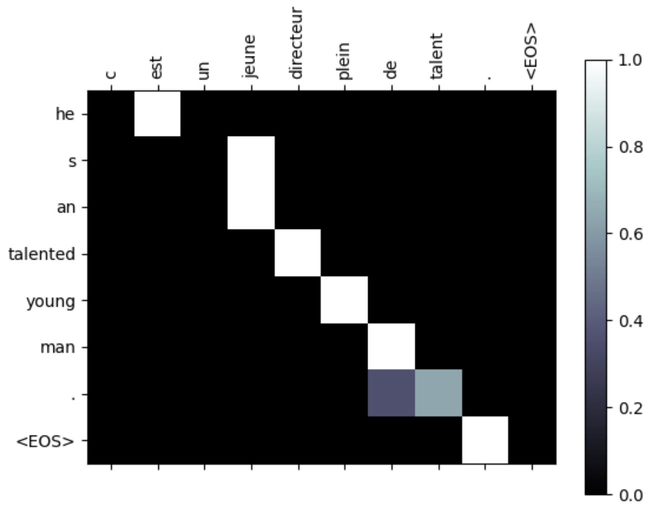
For a etter viewing experience we will o the extra work of a ing axes an la els:

```
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 .")
```









Out:

```
input = elle a cinq ans de moins que moi .
output = she is five 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 afraid to die . <EOS>
input = c est un jeune directeur plein de talent .
output = he s an talented young man . <EOS>
```

Exercises

- Try with a ifferent ataset
  - Another language pair
  - Human → Machine (e.g. IOT comman s)
  - Chat → Response
  - Question → Answer
- Replace the em e ings with pre-traine wor em e ings such as wor 2vec or GloVe
- Try with more layers, more hi en units, an more sentences. Compare the training time an 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 autoenco er. Try this:
  - Train as an autoenco er
  - Save only the Enco er network
  - Train a new Deco er for translation from there

Total running time of the scri t: (21 minutes 41.623 secon s)

Rate this Tutorial