

TRANSLATION WITH A SEQUENCE TO SEQUENCE NETWORK AND ATTENTION

- Translate from French to English
- Sequence to sequence network
 - : two RNN work together to transform one sequence to another.
 - Encoder: condenses an input sequence into a vector
 - Decoder: unfolds the vector into a new sequence

Requirements

```
from __future__ import unicode_literals, print_function, division
from io import open
import unicodedata
import string
import re
import random

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

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

Loading data files

- English to French pairs
- unique index per word & one-hot vector

SOS	EOS	the	a	is	and	or
01	02	03	04	05	06	07 ..

and = $\langle 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \dots \rangle$

- class Lang: helper class
 - word \rightarrow index (word2index)
 - index \rightarrow word (index2word)
 - Count of each word (word2count)

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
```

To simplify:

- turn Unicode characters to ASCII
- make everything lowercase
- trim most punctuation

```
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 :

- split the file into lines
- ↓
- split lines into pairs

To translate from other language → English : reverse flag

```
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
```

To train quickly:

- trim the dataset to only relatively short & simple sentences.
- maximum length: 10 words (includes ending punctuation)
- filtering to sentences that translate to the form "I am" or "He is" etc.

MAX_LENGTH = 10

```
eng_prefixes = (  
    "i am", "i am", "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)]
```

Full process for preparing the data is :

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

```
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))
```

The Seq2Seq Model

- RNN (Recurrent Neural Network)

: a network that operates on a sequence and uses its own output as input for subsequent steps.

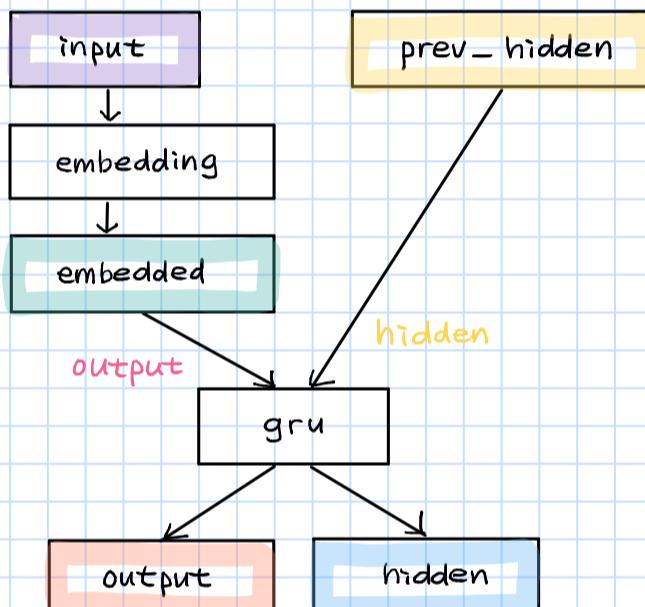
- Encoder: reads an input sequence

↓
outputs a single vector

- Decoder: reads the vector to produce an output sequence

The Encoder

- 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
- uses the hidden state for the next input words



```
class EncoderRNN(nn.Module):
```

```
    def __init__(self, input_size, hidden_size):
        super(EncoderRNN, self).__init__()
        self.hidden_size = hidden_size

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

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

```
    def initHidden(self):
```

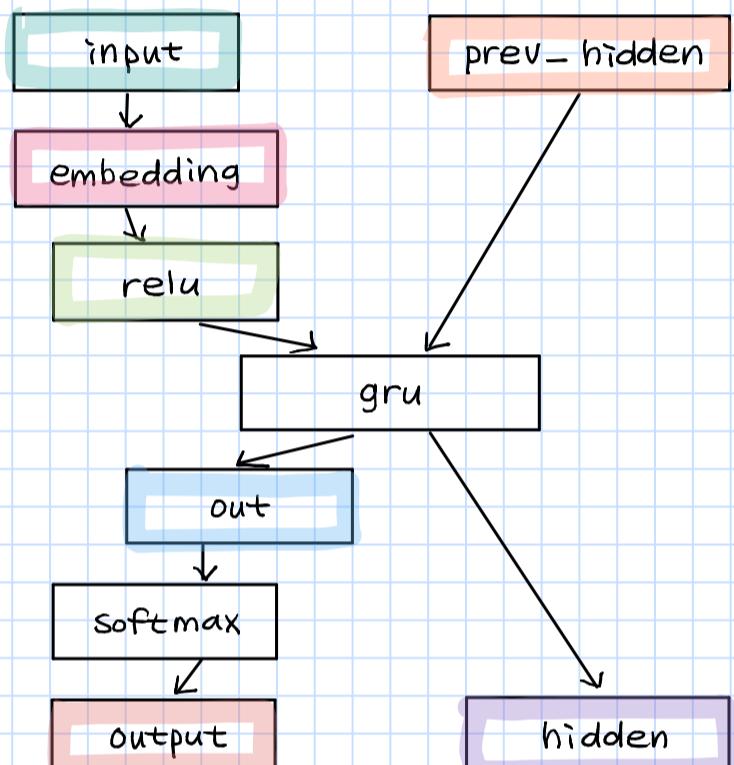
```
        return torch.zeros(1, 1, self.hidden_size, device=device)
```

The Decoder

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

Simple Decoder

- use only last output of the encoder.
sometimes called "context vector"
- Context Vector is used as the initial hidden state of the decoder.
- At every step of decoding, the decoder is given an input token & hidden state.
 - initial input token: start-of-string <SOS>
 - first hidden state: context vector (the encoder's last hidden state)

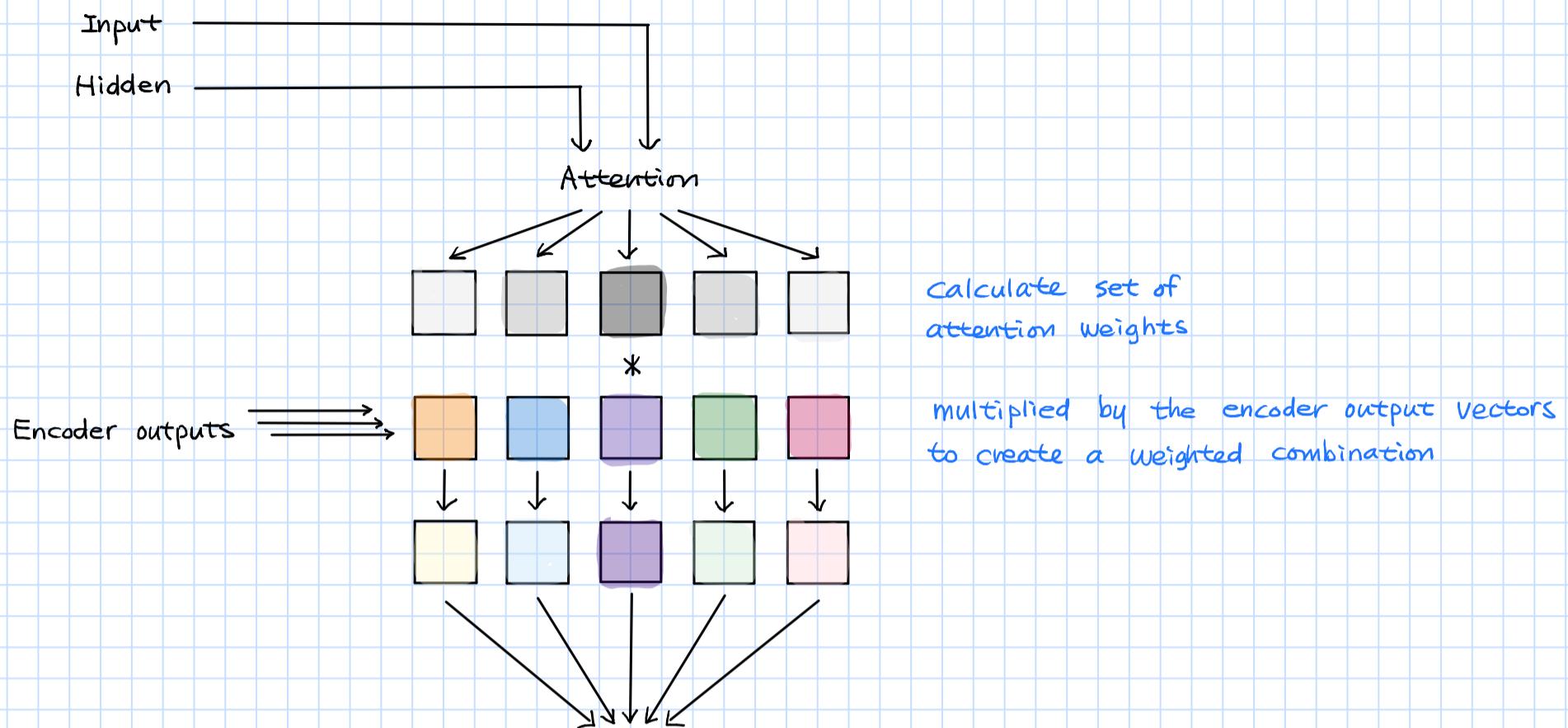


Class DecoderRNN (nn.Module):

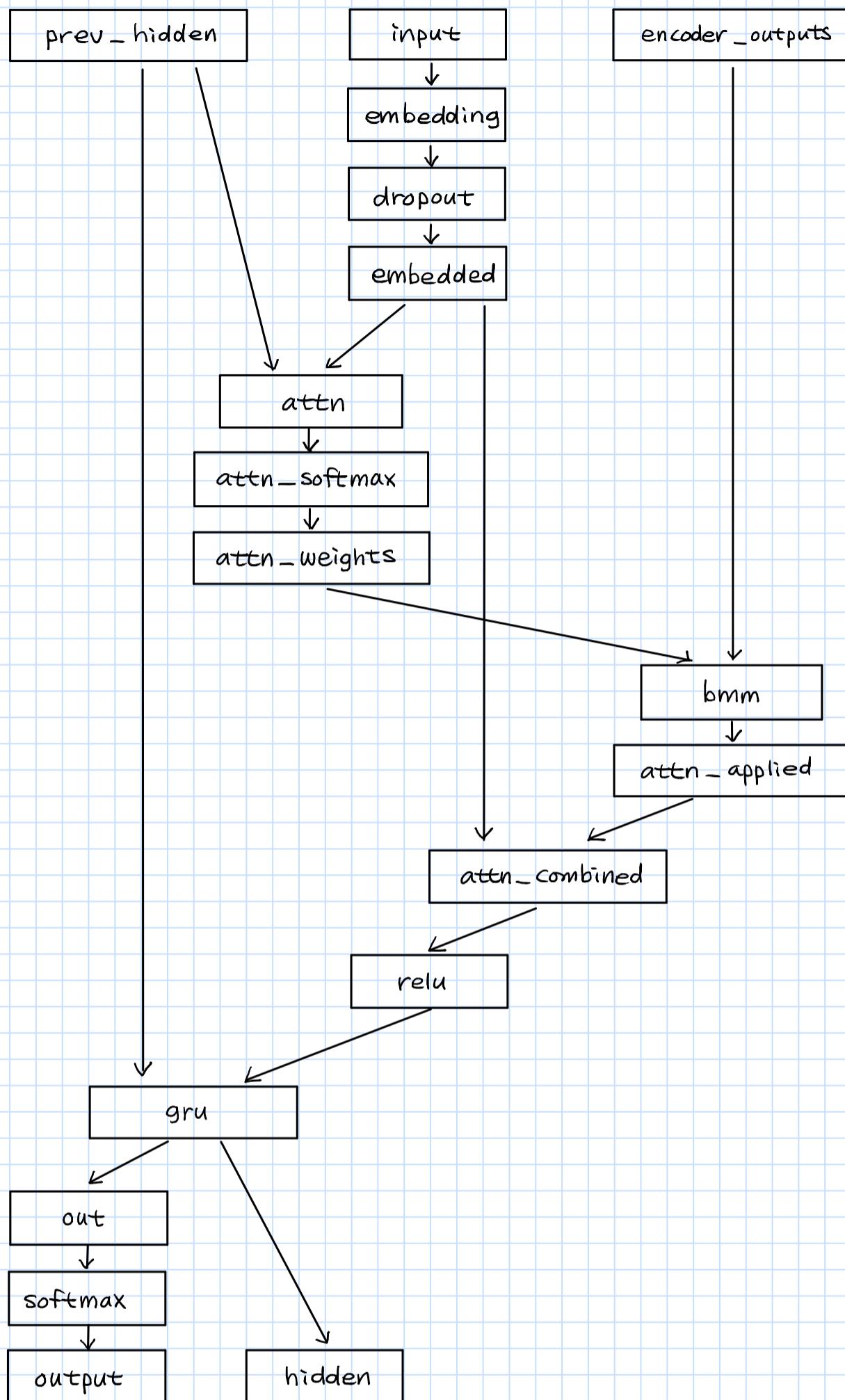
```
def __init__(self, hidden_size, output_size):  
    super(DecoderRNN, self).__init__()  
    self.hidden_size = hidden_size  
  
    self.embedding = nn.Embedding(output_size, hidden_size)  
    self.gru = nn.GRU(hidden_size, hidden_size)  
    self.out = nn.Linear(hidden_size, output_size)  
    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

- 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.



- the result should contain information about the specific part of the input sequence
→ help the decoder choose the right output words



- have to choose a maximum sentence length
- sentences of maximum length will use all the attention weights
 - shorter sentences will only use the first few.

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)
```

Training

Preparing Training Data

- each pair, we will need an input tensor (indexes of the words in the input sentence)
 - & target tensor (indexes of the words in the target sentence)
- while creating these vectors, we will append the EOS token to both sequences.

```
def indexesFromSentences(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

- Encoder

- run the input sentence
- keep track of every output & latest hidden state.

- Decoder

- given the <sos> token as its first input
- last hidden state of encoder as its first hidden state.

- Teacher forcing

: the concept of using real target outputs as each next input,
instead of using the decoder's guess as the next input.

↓

- teacher-forced networks: coherent grammar, but not correct translation

(why?) it has learned to represent the output grammar &
can "pick up" the meaning once the teacher tells it the first few words.
it has not properly learned how to create the sentence from the translation

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 :
            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
```

```
# helper function to print time elapsed and
```

```
# estimated time remaining given the current time and 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)      → 전체 걸릴 시간
```

```
    rs = es - s            → 남은 시간
```

```
    return '%s (- %s)' % (asMinutes(s), asMinutes(rs))
```

Whole training process

1. Start a timer
2. Initialize optimizer & criterion
3. Create set of training pairs
4. Start empty losses array for plotting.

call train many times!

```
def trainIter (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 = [tensorFromPair (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 ('%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

```
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

- no target!
- simply feed the decoder's predictions back to itself for each step.
- word predicted \Rightarrow added to output string
- EOS token \Rightarrow stop there.
- store decoder's attention outputs for display 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)  
            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]
```

*1 NLL Loss (Negative Log-Likelihood)

- output node 와 우리가 가진 레이터 분포 사이의 차이: 크로스 엔트로피
- 이 차이만큼을 loss로 보고 이 loss에 대한 gradient를 구해 예제파하는 것이 "드롭"