Imports

```
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
import numpy as np

%matplotlib inline

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

```
device
```

```
    device(type='cpu')
```

Loading data files

```
%%capture
!wget https://download.pytorch.org/tutorial/data.zip
!unzip -o data.zip
```

Vocabulary Class (and Text Preprocessing)

```
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


# 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

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


MAX_LENGTH = 10

eng_prefixes = (
    "i am ", "i m ",
    "he is", "he s ",
    "she is", "she s ",
    "you are", "you 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)]


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)
```

```
Reading lines...
Read 135842 sentence pairs
Trimmed to 10599 sentence pairs
Counting words...
Counted words:
fra 4345
eng 2803
```

Random Sample for Subsequent Runs

A random sample from the 'pairs' list was chosen so that the maximum length of the input and output sequences do not change upon subsequent runs, since we are not allowed to use a for-loop in our code!

```
# sample = random.choice(pairs)
sample = ['vous me faites rougir .', 'you re making me blush .']
sample
```

```
['vous me faites rougir .', 'you re making me blush .']
```

```
input_sentence = sample[0]
output_sentence = sample[1]

input_indices = [input_lang.word2index[word] for word in input_sentence.split(' ')
target_indices = [output_lang.word2index[word] for word in output_sentence.split('

input_indices.append(EOS_token)
target_indices.append(EOS_token)

input_indices, target_indices
```

```
([118, 27, 590, 2795, 5, 1], [129, 78, 505, 343, 1655, 4, 1])
```

```
input_tensor = torch.tensor(input_indices, dtype=torch.long, device = device)
output_tensor = torch.tensor(target_indices, dtype=torch.long, device = device)

input_tensor.shape, output_tensor.shape
```

```
(torch.Size([6]), torch.Size([7]))
```

Dimensions

```
DIM_IN = input_lang.n_words
DIM_OUT = output_lang.n_words
DIM_HID = 256 # arbitraily chosen! must be same for encoder and decoder!
MAX_LEN_IN = input_tensor.size()[0] # length of the input sequence under considera
MAX_LEN_OUT = output_tensor.size()[0] # length of the output sequence under consid

DIM_IN, DIM_OUT, DIM_HID, MAX_LEN_IN, MAX_LEN_OUT
```

```
(4345, 2803, 256, 6, 7)
```

Encoder

Instantiating layers

```
embedding = nn.Embedding(DIM_IN, DIM_HID).to(device)
lstm = nn.LSTM(DIM_HID, DIM_HID).to(device)
```

Feeding Input Sequence to Encoder

```
encoder_outputs = torch.zeros(MAX_LEN_IN, DIM_HID, device=device) # array to store
hidden = torch.zeros(1, 1, DIM_HID, device=device) # first hidden state initialize
cell = torch.zeros(1, 1, DIM_HID, device=device) # first hidden state initialized


input = input_tensor[0].view(-1, 1)
embedded_input = embedding(input)
output, (hidden, cell) = lstm(embedded_input, (hidden, cell))
encoder_outputs[0] += output[0,0]


input = input_tensor[1].view(-1, 1)
embedded_input = embedding(input)
output, (hidden, cell) = lstm(embedded_input, (hidden, cell))
encoder_outputs[1] += output[0,0]


input = input_tensor[2].view(-1, 1)
embedded_input = embedding(input)
output, (hidden, cell) = lstm(embedded_input, (hidden, cell))
encoder_outputs[2] += output[0,0]


input = input_tensor[3].view(-1, 1)
embedded_input = embedding(input)
output, (hidden, cell) = lstm(embedded_input, (hidden, cell))
encoder_outputs[3] += output[0,0]


input = input_tensor[4].view(-1, 1)
embedded_input = embedding(input)
output, (hidden, cell) = lstm(embedded_input, (hidden, cell))
encoder_outputs[4] += output[0,0]


input = input_tensor[5].view(-1, 1)
embedded_input = embedding(input)
output, (hidden, cell) = lstm(embedded_input, (hidden, cell))
encoder_outputs[5] += output[0,0]
```

Decoder

Instantiating Layers

```
embedding = nn.Embedding(DIM_OUT, DIM_HID).to(device)
attn_weigts_layer = nn.Linear(DIM_HID * 3, MAX_LEN_IN).to(device) # The output and

lstm_inp = nn.Linear(DIM_HID * 2, DIM_HID).to(device) #this layer takes care of th
lstm = nn.LSTM(DIM_HID, DIM_HID).to(device)

linear_out = nn.Linear(DIM_HID, DIM_OUT).to(device)
```

```
predicted_sentence = []
```

Feeding to the Decoder - Word 1

```
decoder_input = torch.tensor([[SOS_token]], device=device) # We start from the <SO
decoder_hidden = hidden # what we got from the output of the encoder from the last
decoder_cell = cell # what we got from the output of the encoder from the last wor

embedded = embedding(decoder_input)

# This decides the values with which output from the encoder needs weighed!
attn_weigts_layer_input = torch.cat((embedded[0], decoder_hidden[0], cell[0]), 1)
attn_weights = attn_weigts_layer(attn_weigts_layer_input)
attn_weights = F.softmax(attn_weights, dim = 1)

# This calculates the attention values!
attn_applied = torch.bmm(attn_weights.unsqueeze(0), encoder_outputs.unsqueeze(0))

input_to_lstm = lstm_inp(torch.cat((embedded[0], attn_applied[0]), 1))
input_to_lstm = input_to_lstm.unsqueeze(0)
output, (decoder_hidden, decoder_cell) = lstm(input_to_lstm, (decoder_hidden, deco
output = F.relu(output)

output = F.softmax(linear_out(output[0]), dim = 1)
top_value, top_index = output.data.topk(1) # same as using np.argmax
out_word = output_lang.index2word[top_index.item()]
print(out_word)
predicted_sentence.append(out_word)
```

```
    should
```

Feeding to the Decoder - Word 2

```
teacher_forcing_ratio = 0.5
use_teacher_forcing = True if random.random() < teacher_forcing_ratio else False

if use_teacher_forcing:
  decoder_input = torch.tensor([[target_indices[0]]], device=device)
else:
  decoder_input = torch.tensor([[top_index.item()]], device=device)



embedded = embedding(decoder_input)

# This decides the values with which output from the encoder needs weighed!
attn_weigts_layer_input = torch.cat((embedded[0], decoder_hidden[0], cell[0]), 1)
attn_weights = attn_weigts_layer(attn_weigts_layer_input)
attn_weights = F.softmax(attn_weights, dim = 1)
```

```
# This calculates the attention values!
attn_applied = torch.bmm(attn_weights.unsqueeze(0), encoder_outputs.unsqueeze(0))

input_to_lstm = lstm_inp(torch.cat((embedded[0], attn_applied[0]), 1))
input_to_lstm = input_to_lstm.unsqueeze(0)
output, (decoder_hidden, decoder_cell) = lstm(input_to_lstm, (decoder_hidden, deco
output = F.relu(output)

output = F.softmax(linear_out(output[0]), dim = 1)
top_value, top_index = output.data.topk(1) # same as using np.argmax
out_word = output_lang.index2word[top_index.item()]
print(out_word)
predicted_sentence.append(out_word)
```

```
    art
```

Feeding to the Decoder - Word 3

```
teacher_forcing_ratio = 0.5
use_teacher_forcing = True if random.random() < teacher_forcing_ratio else False

if use_teacher_forcing:
  decoder_input = torch.tensor([[target_indices[0]]], device=device)
else:
  decoder_input = torch.tensor([[top_index.item()]], device=device)
```

```
embedded = embedding(decoder_input)

# This decides the values with which output from the encoder needs weighed!
attn_weigts_layer_input = torch.cat((embedded[0], decoder_hidden[0], cell[0]), 1)
attn_weights = attn_weigts_layer(attn_weigts_layer_input)
attn_weights = F.softmax(attn_weights, dim = 1)

# This calculates the attention values!
attn_applied = torch.bmm(attn_weights.unsqueeze(0), encoder_outputs.unsqueeze(0))

input_to_lstm = lstm_inp(torch.cat((embedded[0], attn_applied[0]), 1))
input_to_lstm = input_to_lstm.unsqueeze(0)
output, (decoder_hidden, decoder_cell) = lstm(input_to_lstm, (decoder_hidden, deco
output = F.relu(output)

output = F.softmax(linear_out(output[0]), dim = 1)
top_value, top_index = output.data.topk(1) # same as using np.argmax
out_word = output_lang.index2word[top_index.item()]
print(out_word)
predicted_sentence.append(out_word)
```

```
    beer
```

Feeding to the Decoder - Word 4

```
teacher_forcing_ratio = 0.5
use_teacher_forcing = True if random.random() < teacher_forcing_ratio else False

if use_teacher_forcing:
  decoder_input = torch.tensor([[target_indices[0]]], device=device)
else:
  decoder_input = torch.tensor([[top_index.item()]], device=device)
```

```
embedded = embedding(decoder_input)

# This decides the values with which output from the encoder needs weighed!
attn_weigts_layer_input = torch.cat((embedded[0], decoder_hidden[0], cell[0]), 1)
attn_weights = attn_weigts_layer(attn_weigts_layer_input)
attn_weights = F.softmax(attn_weights, dim = 1)

# This calculates the attention values!
attn_applied = torch.bmm(attn_weights.unsqueeze(0), encoder_outputs.unsqueeze(0))

input_to_lstm = lstm_inp(torch.cat((embedded[0], attn_applied[0]), 1))
input_to_lstm = input_to_lstm.unsqueeze(0)
output, (decoder_hidden, decoder_cell) = lstm(input_to_lstm, (decoder_hidden, deco
output = F.relu(output)

output = F.softmax(linear_out(output[0]), dim = 1)
top_value, top_index = output.data.topk(1) # same as using np.argmax
out_word = output_lang.index2word[top_index.item()]
print(out_word)
predicted_sentence.append(out_word)
```

```
    alone
```

Feeding to the Decoder - Word 5

```
teacher_forcing_ratio = 0.5
use_teacher_forcing = True if random.random() < teacher_forcing_ratio else False

if use_teacher_forcing:
  decoder_input = torch.tensor([[target_indices[0]]], device=device)
else:
  decoder_input = torch.tensor([[top_index.item()]], device=device)
```

```
embedded = embedding(decoder_input)

# This decides the values with which output from the encoder needs weighed!
```

```
attn_weigts_layer_input = torch.cat((embedded[0], decoder_hidden[0], cell[0]), 1)
attn_weights = attn_weigts_layer(attn_weigts_layer_input)
attn_weights = F.softmax(attn_weights, dim = 1)

# This calculates the attention values!
attn_applied = torch.bmm(attn_weights.unsqueeze(0), encoder_outputs.unsqueeze(0))

input_to_lstm = lstm_inp(torch.cat((embedded[0], attn_applied[0]), 1))
input_to_lstm = input_to_lstm.unsqueeze(0)
output, (decoder_hidden, decoder_cell) = lstm(input_to_lstm, (decoder_hidden, deco
output = F.relu(output)

output = F.softmax(linear_out(output[0]), dim = 1)
top_value, top_index = output.data.topk(1) # same as using np.argmax
out_word = output_lang.index2word[top_index.item()]
print(out_word)
predicted_sentence.append(out_word)
```

```
    sketching
```

Feeding to the Decoder - Word 6

```
teacher_forcing_ratio = 0.5
use_teacher_forcing = True if random.random() < teacher_forcing_ratio else False

if use_teacher_forcing:
  decoder_input = torch.tensor([[target_indices[0]]], device=device)
else:
  decoder_input = torch.tensor([[top_index.item()]], device=device)
```

```
embedded = embedding(decoder_input)

# This decides the values with which output from the encoder needs weighed!
attn_weigts_layer_input = torch.cat((embedded[0], decoder_hidden[0], cell[0]), 1)
attn_weights = attn_weigts_layer(attn_weigts_layer_input)
attn_weights = F.softmax(attn_weights, dim = 1)

# This calculates the attention values!
attn_applied = torch.bmm(attn_weights.unsqueeze(0), encoder_outputs.unsqueeze(0))

input_to_lstm = lstm_inp(torch.cat((embedded[0], attn_applied[0]), 1))
input_to_lstm = input_to_lstm.unsqueeze(0)
output, (decoder_hidden, decoder_cell) = lstm(input_to_lstm, (decoder_hidden, deco
output = F.relu(output)

output = F.softmax(linear_out(output[0]), dim = 1)
top_value, top_index = output.data.topk(1) # same as using np.argmax
out_word = output_lang.index2word[top_index.item()]
print(out_word)
predicted_sentence.append(out_word)
```

```
discussing
```

Feeding to the Decoder - Word 7

```
teacher_forcing_ratio = 0.5
use_teacher_forcing = True if random.random() < teacher_forcing_ratio else False

if use_teacher_forcing:
  decoder_input = torch.tensor([[target_indices[0]]], device=device)
else:
  decoder_input = torch.tensor([[top_index.item()]], device=device)



embedded = embedding(decoder_input)

# This decides the values with which output from the encoder needs weighed!
attn_weigts_layer_input = torch.cat((embedded[0], decoder_hidden[0], cell[0]), 1)
attn_weights = attn_weigts_layer(attn_weigts_layer_input)
attn_weights = F.softmax(attn_weights, dim = 1)

# This calculates the attention values!
attn_applied = torch.bmm(attn_weights.unsqueeze(0), encoder_outputs.unsqueeze(0))

input_to_lstm = lstm_inp(torch.cat((embedded[0], attn_applied[0]), 1))
input_to_lstm = input_to_lstm.unsqueeze(0)
output, (decoder_hidden, decoder_cell) = lstm(input_to_lstm, (decoder_hidden, deco
output = F.relu(output)

output = F.softmax(linear_out(output[0]), dim = 1)
top_value, top_index = output.data.topk(1) # same as using np.argmax
out_word = output_lang.index2word[top_index.item()]
print(out_word)
predicted_sentence.append(out_word)
```

```
beer
```

```
predicted_sentence = ' '.join(predicted_sentence)
predicted_sentence
```

```
'should art beer alone sketching discussing beer'
```

✓ 0s completed at 9:55 PM