Seq2Seq Model - Neural Machine Translation

- https://blog.keras.io/a-ten-minute-introduction-to-sequence-to-sequence-learning-in-keras.html
- https://github.com/keras-team/keras/blob/master/examples/lstm_seq2seq.py
- https://machinetalk.org/2019/03/29/neural-machine-translation-with-attention-mechanism/

What is **sequence-to-sequence learning?**

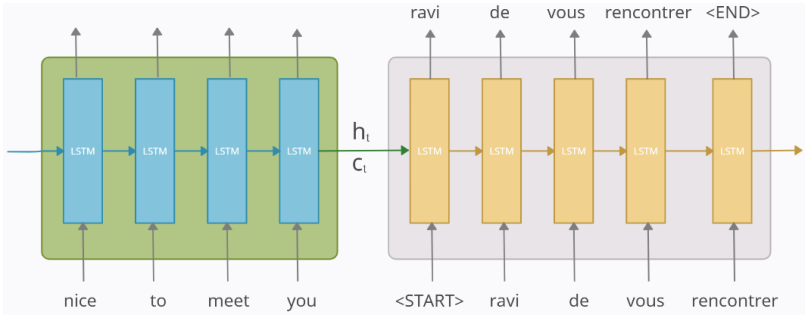
Sequence-to-sequence learning (Seq2Seq) is about training models to convert sequences from one domain (e.g. sentences in English) to sequences in another domain (e.g. the same sentences translated to French).

"the cat sat on the mat" -> [Seq2Seq model] -> "le chat etait assis sur le tapis"

This can be used for machine translation or for free-from question answering (generating a natural language answer given a natural language question) -- in general, it is applicable any time you need to generate text.

The general case: canonical sequence-to-sequence

In the general case, input sequences and output sequences have different lengths (e.g. machine translation) and the entire input sequence is required in order to start predicting the target. This requires a more advanced setup, which is what people commonly refer to when mentioning "sequence to sequence models" with no further context. Here's how it works:



- **A RNN layer (or stack thereof) acts as "encoder": it processes the input sequence and returns its own internal state. The encoder, which is on the left-hand side, requires only sequences from source language as inputs. Note that we discard the outputs of the encoder RNN, only recovering the state. This state will serve as the "context", or "conditioning", of the decoder in the next step.**

- **Another RNN layer (or stack thereof) acts as "decoder": it is trained to predict the next word of the target sequence, given previous words of the target sequence.**
  - **Specifically, it is trained to `turn the target sequences into the same sequences but offset by one timestep in the future` , a training process called `teacher forcing` in this context.**

- **Importantly, `the encoder uses as initial state the state vectors from the encoder`, which is how the decoder obtains information about what it is supposed to generate.**
- **Effectively, the decoder learns to generate target at $t+1$ given target at t, conditioned on the input sequence.**
- **The same process can also be used to train a Seq2Seq network `without teacher forcing`, i.e. by reinjecting the decoder's predictions into the decoder.**

Neural Translation Machine

Let's illustrate these ideas with actual code.

For our example implementation, we will use a dataset of pairs of English sentences and their French translation, which you can download from manythings.org/anki. The file to download is called `fra-eng.zip` (English/French). We will implement a word-level model sequence-to-sequence model, processing the input word-by-word and generating the output word-by-word.

Here's a summary of our process:

- **1. Turn the sentences into 3 Numpy arrays, `encoder_input_data`, `decoder_input_data`, `decoder_target_data`:**
  - **`encoder_input_data` is a 3D array of shape (`num_pairs`, `max_english_sentence_length`, `num_english_characters`) containing a one-hot vectorization of the English sentences.**
  - **`decoder_input_data` is a 3D array of shape (`num_pairs`, `max_french_sentence_length`, `num_french_characters`) containg a one-hot vectorization of the French sentences.**
  - **`decoder_target_data` is the same as decoder_input_data but offset by one timestep. `decoder_target_data[:, t, :]` will be the same as `decoder_input_data[:, t + 1, :]`.**
- **2. Train a basic LSTM-based Seq2Seq model to predict `decoder_target_data` given `encoder_input_data` and `decoder_input_data`. Our model uses `teacher forcing`.**
- **3. Decode some sentences to check if the model is working (i.e. turn samples from `encoder_input_data` into corresponding samples from `decoder_target_data`).**

Because the `training process` and `inference process` (decoding sentences) are quite different, `we use different models for both, albeit they all leverage the same inner layers`.

Note that the encoder and decoder are connected by RNN states:

- **`encoder states`: This is used to store the states of the encoder.**
- **`inital_state of decoder`: we pass the encoder states to the decoder as initial states.**

In [1]:
```
!pip install torchinfo
```

```
Collecting torchinfo
  Downloading torchinfo-1.6.5-py3-none-any.whl (21 kB)
Installing collected packages: torchinfo
Successfully installed torchinfo-1.6.5
```

In [2]:
```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

import numpy as np
import unicodedata
```

```
import re
from tensorflow import keras
import pandas as pd
from sklearn.utils import shuffle
```

(1) Data Preprocessing

1. **Turn the sentences into 3 Numpy arrays, `data_en`, `data_fr_in`, `data_fr_out`:**

- **`data_en`: is a 2D array of shape (`num_samples`, `max_en_words_per_sentence`) containing a tokenized sentences after preprocessing.**
- **`data_fr_in`: is a 2D array of shape (`num_samples`, `max_fr_words_per_sentence`) containing a tokenized sentences after preprocessing.**
- **`data_fr_out`: the same as decoder_input_data but offset by one timestep, i.e. `data_fr_in [:, t]` will be the same as `data_fr_out [:, t+1]`.**

Note that, this is a demo version of the NTM. We will train the model using a small dataset. To make the model work realistically, you need to train the model with a larget collection of training samples

We'll use Keras for simple data preprocessing

In [11]:
```
from google.colab import drive
drive.mount('/content/drive')

# Point to the training data file
path_to_glove_file = "drive/MyDrive/BIA667_Lab/fra.txt"
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/cont
ent/drive", force_remount=True).
```

In [12]:
```
# Read data
#text = pd.read_csv('fra.txt', sep="\t", header=None, usecols=[0,1])
text = pd.read_csv(path_to_glove_file, sep="\t", header=None, usecols=[0,1])
text.columns =['en','fr']
text.head()
len(text)

# Take a small set to save training time
text = shuffle(text)
raw_data=text.iloc[0:10000]
```

Out[12]:

	en	fr
0	Go.	Va !
1	Hi.	Salut !
2	Hi.	Salut.
3	Run!	Cours !
4	Run!	Courez !

Out[12]:
```
175623
```

In [13]:
```
# clean up text

def unicode_to_ascii(s):
```

```
    return ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn')


def normalize_string(s):
    s = unicode_to_ascii(s)
    s = re.sub(r'([!.?])', r' \1', s)
    s = re.sub(r'[^a-zA-Z.!?]+', r' ', s)
    s = re.sub(r'\s+', r' ', s)
    return s
```

In [14]:
```
# clean up text
raw_data_en = [normalize_string(data) for data in raw_data["en"]]

# add special token <start>/<end> to indicate the beginning and end of a sentence
raw_data_fr_in = ['<start> ' + normalize_string(data) for data in raw_data["fr"]]
raw_data_fr_out = [normalize_string(data) + ' <end>' for data in raw_data["fr"]]
```

In [15]:
```
# Tokenize each sentence and index each word
max_en_words = 5000
max_en_len = 10
en_tokenizer = keras.preprocessing.text.Tokenizer(filters='', \
                                                  num_words=max_en_words )
en_tokenizer.fit_on_texts(raw_data_en)
print("Total number of English words: ", len(en_tokenizer.word_index))
```

```
Total number of English words:  4616
```

In [16]:
```
data_en = en_tokenizer.texts_to_sequences(raw_data_en)
data_en = keras.preprocessing.sequence.pad_sequences(data_en,\
                                                     maxlen=max_en_len, \
                                                     padding='post')
# print a sample sentence after preprocessing
print(data_en[:3])
```

```
[[  24  418    4   31  186   52   66  104  516    1]
 [   3   63    8  331    4  549   17    1    0    0]
 [   2   28    8 1380    1    0    0    0    0    0]]
```

In [17]:
```
# Process French sentences in the same way

max_fr_words = 5000
max_fr_len = 10
fr_tokenizer = keras.preprocessing.text.Tokenizer(filters='', num_words = max_fr_words)

# ATTENTION: always finish with fit_on_texts before moving on
fr_tokenizer.fit_on_texts(raw_data_fr_in)
fr_tokenizer.fit_on_texts(raw_data_fr_out)
print("Total number of French words: ", len(fr_tokenizer.word_index))

data_fr_in = fr_tokenizer.texts_to_sequences(raw_data_fr_in)
data_fr_in = keras.preprocessing.sequence.pad_sequences(data_fr_in,\
                                                        maxlen=max_fr_len, \
                                                        padding='post')

data_fr_out = fr_tokenizer.texts_to_sequences(raw_data_fr_out)
data_fr_out = keras.preprocessing.sequence.pad_sequences(data_fr_out,\
                                                         maxlen=max_fr_len, \
                                                         padding='post')
# print a sample sentence after preprocessing
data_fr_in[:3]
```

```
Total number of French words:  6444
```

Out[17]:
```
array([[   5,    5,   35,   63,  115,   50, 1134,   21,  119,    1],
       [   2,   10,  884,   26,  320,   32, 1001,  126,    1,    0],
       [   2,    4,   29, 2029,    1,    0,    0,    0,    0,    0]],
      dtype=int32)
```

In [18]:
```
# Create the reversal mapping between indexes and words
reverse_fr_word_index ={fr_tokenizer.word_index[w] : w \
                        for w in  fr_tokenizer.word_index}
print("index of symbol <start> :", fr_tokenizer.word_index["<start>"])
print("index of symbol <end> :", fr_tokenizer.word_index["<end>"])
```

```
index of symbol <start> : 2
index of symbol <end> : 3
```

(2) Define "Teacher Forcing" Model for Training Process

2. **Train a basic LSTM-based Seq2Seq model to predict `decoder_outputs` given `encoder_inputs` and `decoder_inputs`. Our model uses `teacher forcing`.**

And here is how the data's shape changes at each layer. Often keeping track of the data's shape is extremely helpful not to make silly mistakes, just like stacking up Lego pieces.

Here we start with a simple encoder: only one layer LSTM, unidirectional.

Task for you: `Can you modify the model to allow multiple layers and bidirectional?`

In [19]:
```
import torch
from torch import nn
from torch.utils.data import Dataset, DataLoader, random_split

import torch.optim as optim
```

In [20]:
```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

In [21]:
```
max_en_words = 5000
max_en_len = 10
max_fr_len = 10
max_fr_words = 5000
latent_dim = 100   #i.e. rnn_size
batch_size = 32
```

In [22]:
```
# vocab_size: the total number of words in vocabulary
# embedding_size: word embedding dimension
# latent_dim: RNN hidden state dimension

class EncoderLSTM(nn.Module):

  def __init__(self, vocab_size, embedding_size, hidden_size):

    super(EncoderLSTM, self).__init__()

    self.vocab_size = vocab_size

    self.embedding_size = embedding_size

    self.hidden_size = hidden_size
```

```
    self.embedding = nn.Embedding(self.vocab_size, self.embedding_size,padding_idx=0)

    # The input to LSTM should be [batch_size, seq_length, embedding_size]
    self.LSTM = nn.LSTM(input_size = self.embedding_size, \
                        hidden_size = self.hidden_size,
                        batch_first = True)


  def forward(self, x):

    # the shape of x is [batch_size, seq_length]
    x = self.embedding(x)
    # after embeding, the shape of x is: [batch_size, seq_length, embedding_size]

    # We don't care about output. We only need states
    outputs, (hidden_state, cell_state) = self.LSTM(x)
    # hidden_state shape: [1, batch_size, hidden_size]
    # cell_state shape: [1, batch_size, hidden_size]

    #print(hidden_state.shape)
    #print(cell_state.shape)

    return outputs, hidden_state, cell_state
```

In [23]:
```
encoder = EncoderLSTM(vocab_size=max_en_words,
                      embedding_size=latent_dim,
                      hidden_size=latent_dim)

from torchinfo import summary
summary(encoder,input_size=(batch_size,max_en_len),
       dtypes=[torch.long])
```

Out[23]:
```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
EncoderLSTM                              --                        --
├─Embedding: 1-1                         [32, 10, 100]             500,000
├─LSTM: 1-2                              [32, 10, 100]             80,800
==========================================================================================
Total params: 580,800
Trainable params: 580,800
Non-trainable params: 0
Total mult-adds (M): 41.86
==========================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 0.51
Params size (MB): 2.32
Estimated Total Size (MB): 2.84
==========================================================================================
```

In [24]:
```
# vocab_size: the total number of words in vocabulary
# embedding_size: word embedding dimension
# latent_dim: RNN hidden state dimension

class DecoderLSTM(nn.Module):

  def __init__(self, vocab_size, embedding_size, hidden_size):

    super(DecoderLSTM, self).__init__()

    self.vocab_size = vocab_size
```

```
    self.embedding_size = embedding_size

    self.hidden_size = hidden_size

    self.embedding = nn.Embedding(self.vocab_size, self.embedding_size,padding_idx=0)

    # The input to LSTM should be [batch_size, seq_length, embedding_size]
    self.LSTM = nn.LSTM(input_size = self.embedding_size, \
                        hidden_size = self.hidden_size,
                        batch_first = True)

    self.dense = nn.Linear(in_features = self.hidden_size,
                        out_features = self.vocab_size)


  def forward(self, x, hidden_state, cell_state):


    x = self.embedding(x)

    # LSTM will be initialized with encoder states
    outputs, (h, c) = self.LSTM(x, (hidden_state, cell_state))
    # outputs shape is [batch, seq_length, hidden_state]

    predictions = self.dense(outputs)
    # prediction shape is [batch, seq_length, vocab_size]

    return predictions, h, c
```

In [25]:
```
decoder = DecoderLSTM(vocab_size=max_fr_words,\
                      embedding_size=latent_dim,\
                      hidden_size=latent_dim)

summary(decoder,input_size=[(batch_size,max_fr_len),\
                            (1,batch_size,latent_dim),\
                            (1,batch_size,latent_dim)],\
        dtypes=[torch.long,torch.float,torch.float])
```

Out[25]:
```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
DecoderLSTM                              --                        --
├─Embedding: 1-1                         [32, 10, 100]             500,000
├─LSTM: 1-2                              [32, 10, 100]             80,800
├─Linear: 1-3                            [32, 10, 5000]            505,000
==========================================================================================
Total params: 1,085,800
Trainable params: 1,085,800
Non-trainable params: 0
Total mult-adds (M): 58.02
==========================================================================================
Input size (MB): 0.03
Forward/backward pass size (MB): 13.31
Params size (MB): 4.34
Estimated Total Size (MB): 17.68
==========================================================================================
```

Connect Encoder and Decoder to Create Seq2Seq Model

In [26]:
```
class Seq2Seq(nn.Module):
```

```
  def __init__(self, Encoder_LSTM, Decoder_LSTM):

    super(Seq2Seq, self).__init__()

    self.Encoder = Encoder_LSTM
    self.Decoder = Decoder_LSTM

  def forward(self, encoder_input, decoder_input):

    _, hidden_state, cell_state = self.Encoder(encoder_input)

    predictions, _, _ = self.Decoder(decoder_input, hidden_state, cell_state)

    return predictions
```

In [27]:
```
model = Seq2Seq(encoder, decoder)
```

Create Dataset and Training Function

Now you can compile and fit the model as usual. Note the matching between tensors and variables:

- `encoder_inputs` <-> `data_en`
- `decoder_inputs` <-> `data_fr_in`
- `decoder_outputs` <-> `data_fr_out`

In [28]:
```
class NTM_dataset(Dataset):

    def __init__(self,data_en, data_fr_in,data_fr_out):

        self.length = len(data_en)

        self.encoder_input = torch.IntTensor(data_en)
        self.decoder_input = torch.IntTensor(data_fr_in)

        # for CrossEntropyLoss, decoder_output must have Long data type
        self.decoder_output = torch.LongTensor(data_fr_out)

    def __getitem__(self, index):
        return self.encoder_input[index], \
               self.decoder_input[index],\
               self.decoder_output[index]

    def __len__(self):
        return self.length
```

In [29]:
```
dataset = NTM_dataset(data_en, data_fr_in, data_fr_out)

test_size = int(len(data_en) * 0.2)
train_size = len(data_en) - test_size

train_dataset, test_dataset = torch.utils.data.random_split(dataset, \
                                                            [train_size, test_size])
```

In [31]:
```
print(len(train_dataset))
len(test_dataset)
```

```
8000
```

Out[31]:
```
2000
```

In [32]:

```
# Define a function to train the model
def train_model(model, train_dataset, test_dataset, device, lr=0.0005, epochs=20, batch_size=32)

    # construct dataloader
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
    test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=True)

    # move model to device
    model = model.to(device)

    # history
    history = {'train_loss': [],
               'train_acc': [],
               'test_loss': [],
               'test_acc': []}
    # setup loss function and optimizer
    optimizer = torch.optim.RMSprop(model.parameters(), lr=lr)
    criterion = nn.CrossEntropyLoss()

    # training loop
    print('Training Start')
    for epoch in range(epochs):
        model.train()
        train_loss = 0
        train_acc = 0
        test_loss = 0
        test_acc = 0

        for encoder_input, decoder_input, decoder_output in train_loader:

            # move data to device
            encoder_input = encoder_input.to(device)
            decoder_input = decoder_input.to(device)
            decoder_output = decoder_output.to(device)

            # forward
            outputs = model(encoder_input, decoder_input)  # batch_size, max_fr_len (i.e. seq_le

            _, pred = torch.max(outputs, dim = -1)

            #reshape output to batch_size * seq_len, fr_vocab_size since the loss looks for 2-di
            cur_train_loss = criterion(outputs.view(-1 , max_fr_words), decoder_output.view(-1))

            # reshape pred & decoder ouput t to calculate acc of each predicted words
            cur_train_acc = (pred.view(-1) == decoder_output.view(-1))
            cur_train_acc = cur_train_acc.sum().item()/len(cur_train_acc)

            # backward
            cur_train_loss.backward()
            optimizer.step()
            optimizer.zero_grad()
            # loss and acc
            train_loss += cur_train_loss
            train_acc += cur_train_acc

        # test start
        model.eval()
        with torch.no_grad():

            for encoder_input, decoder_input, decoder_output in test_loader:
```

```
                # move data to device
                encoder_input = encoder_input.to(device)
                decoder_input = decoder_input.to(device)
                decoder_output = decoder_output.to(device)

                # forward
                outputs = model(encoder_input, decoder_input)  # batch_size, max_fr_len (i.e. se

                _, pred = torch.max(outputs, dim = -1)

                #reshape output to batch_size, seq_len * fr_vocab_size since the loss looks for
                cur_test_loss = criterion(outputs.view(-1, max_fr_words), decoder_output.view(-1

                cur_test_acc = (pred.view(-1) == decoder_output.view(-1))
                cur_test_acc = cur_test_acc.sum().item()/len(cur_test_acc)

                # loss and acc
                test_loss += cur_test_loss
                test_acc += cur_test_acc

        # epoch output
        train_loss = (train_loss/len(train_loader)).item()
        train_acc = train_acc/len(train_loader)
        val_loss = (test_loss/len(test_loader)).item()
        val_acc = test_acc/len(test_loader)
        history['train_loss'].append(train_loss)
        history['train_acc'].append(train_acc)
        history['test_loss'].append(val_loss)
        history['test_acc'].append(val_acc)
        print(f"Epoch:{epoch + 1} / {epochs}, train loss:{train_loss:.4f} train_acc:{train_acc:.

    return history
```

In [33]:

```
history = train_model(model=model,
                      train_dataset = train_dataset,
                      test_dataset = test_dataset,
                      device=device,
                      epochs=50,
                      batch_size=64)
```

```
Training Start
Epoch:1 / 50, train loss:4.5531 train_acc:0.3292, valid loss:4.0302 valid acc:0.4017
Epoch:2 / 50, train loss:3.8761 train_acc:0.4113, valid loss:3.8225 valid acc:0.4244
Epoch:3 / 50, train loss:3.6659 train_acc:0.4326, valid loss:3.6454 valid acc:0.4452
Epoch:4 / 50, train loss:3.4935 train_acc:0.4547, valid loss:3.5324 valid acc:0.4617
Epoch:5 / 50, train loss:3.3514 train_acc:0.4684, valid loss:3.4392 valid acc:0.4745
Epoch:6 / 50, train loss:3.2308 train_acc:0.4792, valid loss:3.3732 valid acc:0.4788
Epoch:7 / 50, train loss:3.1239 train_acc:0.4876, valid loss:3.3045 valid acc:0.4895
Epoch:8 / 50, train loss:3.0282 train_acc:0.4953, valid loss:3.2686 valid acc:0.4915
Epoch:9 / 50, train loss:2.9401 train_acc:0.5028, valid loss:3.2083 valid acc:0.4990
Epoch:10 / 50, train loss:2.8579 train_acc:0.5095, valid loss:3.1691 valid acc:0.5056
Epoch:11 / 50, train loss:2.7828 train_acc:0.5159, valid loss:3.1441 valid acc:0.5090
Epoch:12 / 50, train loss:2.7100 train_acc:0.5218, valid loss:3.1253 valid acc:0.5136
Epoch:13 / 50, train loss:2.6434 train_acc:0.5280, valid loss:3.0947 valid acc:0.5159
Epoch:14 / 50, train loss:2.5794 train_acc:0.5335, valid loss:3.0711 valid acc:0.5185
Epoch:15 / 50, train loss:2.5170 train_acc:0.5390, valid loss:3.0439 valid acc:0.5206
Epoch:16 / 50, train loss:2.4581 train_acc:0.5447, valid loss:3.0408 valid acc:0.5239
Epoch:17 / 50, train loss:2.4029 train_acc:0.5499, valid loss:3.0162 valid acc:0.5249
Epoch:18 / 50, train loss:2.3483 train_acc:0.5557, valid loss:2.9895 valid acc:0.5298
Epoch:19 / 50, train loss:2.2977 train_acc:0.5602, valid loss:3.0147 valid acc:0.5287
Epoch:20 / 50, train loss:2.2465 train_acc:0.5662, valid loss:2.9902 valid acc:0.5307
Epoch:21 / 50, train loss:2.1976 train_acc:0.5704, valid loss:2.9681 valid acc:0.5342
Epoch:22 / 50, train loss:2.1513 train_acc:0.5753, valid loss:2.9636 valid acc:0.5345
Epoch:23 / 50, train loss:2.1056 train_acc:0.5812, valid loss:2.9655 valid acc:0.5345
Epoch:24 / 50, train loss:2.0607 train_acc:0.5852, valid loss:2.9499 valid acc:0.5379
Epoch:25 / 50, train loss:2.0178 train_acc:0.5903, valid loss:2.9585 valid acc:0.5363
Epoch:26 / 50, train loss:1.9750 train_acc:0.5953, valid loss:2.9544 valid acc:0.5360
Epoch:27 / 50, train loss:1.9349 train_acc:0.6007, valid loss:2.9348 valid acc:0.5379
Epoch:28 / 50, train loss:1.8938 train_acc:0.6061, valid loss:2.9374 valid acc:0.5403
Epoch:29 / 50, train loss:1.8537 train_acc:0.6124, valid loss:2.9557 valid acc:0.5379
Epoch:30 / 50, train loss:1.8175 train_acc:0.6165, valid loss:2.9421 valid acc:0.5412
Epoch:31 / 50, train loss:1.7782 train_acc:0.6219, valid loss:2.9345 valid acc:0.5403
Epoch:32 / 50, train loss:1.7408 train_acc:0.6271, valid loss:2.9408 valid acc:0.5376
Epoch:33 / 50, train loss:1.7071 train_acc:0.6324, valid loss:2.9594 valid acc:0.5399
Epoch:34 / 50, train loss:1.6701 train_acc:0.6394, valid loss:2.9587 valid acc:0.5396
Epoch:35 / 50, train loss:1.6373 train_acc:0.6449, valid loss:2.9474 valid acc:0.5399
Epoch:36 / 50, train loss:1.6021 train_acc:0.6504, valid loss:2.9681 valid acc:0.5367
Epoch:37 / 50, train loss:1.5697 train_acc:0.6561, valid loss:2.9575 valid acc:0.5387
Epoch:38 / 50, train loss:1.5372 train_acc:0.6627, valid loss:2.9661 valid acc:0.5383
Epoch:39 / 50, train loss:1.5052 train_acc:0.6690, valid loss:2.9667 valid acc:0.5398
Epoch:40 / 50, train loss:1.4735 train_acc:0.6762, valid loss:2.9656 valid acc:0.5418
Epoch:41 / 50, train loss:1.4434 train_acc:0.6816, valid loss:2.9758 valid acc:0.5395
Epoch:42 / 50, train loss:1.4131 train_acc:0.6871, valid loss:2.9868 valid acc:0.5396
Epoch:43 / 50, train loss:1.3845 train_acc:0.6939, valid loss:2.9931 valid acc:0.5407
Epoch:44 / 50, train loss:1.3536 train_acc:0.7012, valid loss:3.0008 valid acc:0.5384
Epoch:45 / 50, train loss:1.3259 train_acc:0.7065, valid loss:3.0062 valid acc:0.5405
Epoch:46 / 50, train loss:1.2981 train_acc:0.7131, valid loss:3.0162 valid acc:0.5420
Epoch:47 / 50, train loss:1.2707 train_acc:0.7192, valid loss:3.0181 valid acc:0.5380
Epoch:48 / 50, train loss:1.2446 train_acc:0.7251, valid loss:3.0168 valid acc:0.5403
Epoch:49 / 50, train loss:1.2185 train_acc:0.7307, valid loss:3.0374 valid acc:0.5406
Epoch:50 / 50, train loss:1.1916 train_acc:0.7371, valid loss:3.0506 valid acc:0.5402
```

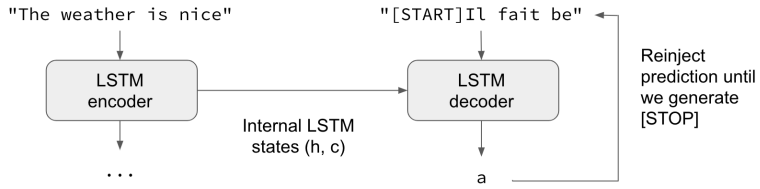
Define a model for Inference (i.e. Testing Translation)

3. **Decode some sentences to check that the model is working (i.e. turn samples from `data_en` into corresponding samples from `data_fr_out`).**

Because the training process and `inference process` (decoding sentences) are quite different, we use different models for both, albeit they all leverage the same inner layers where all the weights

have been trained.

Inference without "teacher forcing"



Now we define a function which returns the translated sentences given an input sentence.

- **The decoder translates word by word starting with an decoder input `[ <start> ]`**
- **When a word, say w_t is translated, the next decode input at $t+1$ is $[w_t]$**
- **It continues until either `<end>` is generated or the `max_fr_len` number of words are generated.**

In [41]:
```
def decode_sequence(model, input_seq, device):  # input_seq is a English sentence

    model.eval()

    with torch.no_grad():
        # convert input_seq to tensor
        input_seq = torch.IntTensor(input_seq).to(device)

        # Encode the input as state vectors.
        _, hidden_state, cell_state = model.Encoder(input_seq)


        # Generate empty target sequence of (batch=1, length=1). (i.e. we translate word by word)
        target_seq = torch.empty((1,1), dtype=torch.int32, device = device)

        # Populate the start symbol of target sequence with the start character.
        target_seq = target_seq.fill_(fr_tokenizer.word_index["<start>"])

        target_seq = target_seq.to(device)

        # Generate word by word using the encode state and the last
        # generated word

        decoded_sentence = []

        while True:

            # get decode ouput and hidden states, output shape is [1,1,5000]
            output_tokens, h, c = model.Decoder(target_seq, hidden_state, cell_state)


            # Get the most likely word
            _, sampled_token_index = torch.max(output_tokens, dim = -1)

            # flatten the token_index and convert to numpy number
            sampled_token_index = sampled_token_index.view(-1).item()

            # Look up the word by id
            sampled_word = reverse_fr_word_index[sampled_token_index]
```

```
            # append the word to decoded sentence
            decoded_sentence.append(sampled_word)

            # Exit condition: either hit max length
            # or find stop character.
            if (sampled_word == '<end>' or len(decoded_sentence) == max_fr_len):
                break

            # Update the target sequence with newly generated word.
            target_seq = target_seq.fill_(sampled_token_index)

            # Update states
            hidden_state, cell_state = h, c

    return ' '.join(decoded_sentence)
```

In [35]:
```
# Now let's test

test = text.iloc[10000:10010]
test
```

Out[35]:

	en	fr
53367	He can't know the truth.	Il ne peut pas connaître la vérité.
109320	I like hot tea better than cold.	Je préfère le thé chaud que froid.
36625	Let's go out tonight?	On sort ce soir ?
94947	Do you care to hazard a guess?	Voudriez-vous hasarder une hypothèse?
95368	Have you gone over the lesson?	As-tu révisé la leçon?
51758	Who said I was ashamed?	Qui a dit que j'avais honte ?
111860	Tom has to call his grandmother.	Tom doit appeler sa grand-mère.
3746	I started it.	Je l'ai initié.
66709	Don't be cruel to animals.	Ne sois pas cruel envers les animaux.
44934	What did you just say?	Que viens-tu de dire ?

In [36]:
```
# preprocess test data
data_test = en_tokenizer.texts_to_sequences(test["en"])
data_test = keras.preprocessing.sequence.pad_sequences(data_test,\
                                                       maxlen=max_en_len, \
                                                       padding='post')
print(data_test[:3])
```

```
[[ 13  36   5   0   0   0   0   0   0   0]
 [  2  38 360 600 161 105   0   0   0   0]
 [ 45  70   0   0   0   0   0   0   0   0]]
```

In [42]:
```
for i in range(len(data_test)):
    #data_test[i].shape
    fr = decode_sequence(model, data_test[i][None,:], device)
    print("\nEn: ", test.iloc[i]["en"])
    print("Fr: ", test.iloc[i]["fr"])
    print("Translated: ", fr)
```

```
En:  He can't know the truth.
Fr:  Il ne peut pas connaître la vérité.
Translated:  il est la reception de ton pere ? <end>

En:  I like hot tea better than cold.
Fr:  Je préfère le thé chaud que froid.
Translated:  ce qui est plus jeune d habitude d aller en

En:  Let's go out tonight?
Fr:  On sort ce soir ?
Translated:  a qui s il vous plait ? <end>

En:  Do you care to hazard a guess?
Fr:  Voudriez-vous hasarder une hypothèse ?
Translated:  voulez vous un peu de the ? <end>

En:  Have you gone over the lesson?
Fr:  As-tu révisé la leçon?
Translated:  tu as la lumiere le francais ? <end>

En:  Who said I was ashamed?
Fr:  Qui a dit que j'avais honte ?
Translated:  il etait deja fait qui etait trop tard . <end>

En:  Tom has to call his grandmother.
Fr:  Tom doit appeler sa grand-mère.
Translated:  tom a t il a la maison . <end>

En:  I started it.
Fr:  Je l'ai initié.
Translated:  j aime les deux questions et mary . <end>

En:  Don't be cruel to animals.
Fr:  Ne sois pas cruel envers les animaux.
Translated:  comment prenez les choses au travail ? <end>

En:  What did you just say?
Fr:  Que viens-tu de dire ?
Translated:  que tu es en train de temps de boire ?
```

Seq2Seq model with attention

Now, let's talk about attention mechanism. What is it and why do we need it?

- **Difficult to remember and process long complicated context**
- **Struggle with difference in syntax structures used by languages**

Implementation of a variety of Seq-2-Seq model can be found here: https://github.com/bentrevett/pytorch-seq2seq

There are different ways to implement such an attention mechanism. You can find the Torch implemetation at https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html

Take aways

- **Neural Machine Translation contains an encoder and a decoder which both are LSTM layers**
- **Attention mechanism can help align encoder and decoder outputs**

- **You can try the following steps to enhance the translation models:**
  - **Use bidirectional LSTM**
  - **Try other more advanced attention mechanisms**
  - **Also, you may need to work on masking when allocating attention scores.**