Import Necessary Libraries

In [2]:
```
# Preprocessing
from __future__ import unicode_literals, print_function, division
from io import open
import unicodedata
import re
import random

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

# Numpy + Matrix Operations
import numpy as np
from torch.utils.data import TensorDataset, DataLoader, RandomSampler

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

Mount Google Drive

Mount Drive to access dataset.

Dataset - Many Things (English - German)

URL - https://www.manythings.org/anki/

In [3]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Preprocessing

There are two main objectives of the preproccessing.

1: Language Encoding

Encode the language using one-hot vectors [0, 0, 0, 1, 0, ... ,0] to access each unique word in the language by using a word-to-index dictionary. Additionally we will track the unique word count as well as each word's frequency.

2: File Text Processing

We will split the file by the tabulation character '\t' to seperate the English, German and CC. We will then use regex to eliminate the CC tags and created a nested list of Enlgish and German sentence pairs.

ex. [[English Sentence, German Sentence], ...]

In [4]:
```
SOS_Token = 0
EOS_Token = 1

class Language:
  def __init__(self, name):
    self.name = name # language
    self.word_to_idx = {} # word to index mapping
    self.idx_to_word = {0: "SOS", 1: "EOS"} # index to word mapping
    self.word_freq = {} # word to frequency mapping
    self.word_count = 2 # unique word count

  def add_sentence(self, sentence):
    for word in sentence.split(' '): # for each word (split by whitespace
      self.add_word(word) # use add word method

  def add_word(self, word):
    # New Word
    if word not in self.word_to_idx:
      self.word_to_idx[word] = self.word_count # add word as key -> count
      self.idx_to_word[self.word_count] = word # add count as key -> word
      self.word_freq[word] = 1 # initialize frequency
      self.word_count += 1 # increment unique word count
    # Repeated Word
    else:
      self.word_freq[word] += 1 # increment word frequency
```

Unicode to ASCII

Since these databases are multilingual there are in unicode not ASCII so using these methods to convert.

In [5]:
```
def unicode_to_ASCII(string):
  return ''.join(
      c for c in unicodedata.normalize('NFD', string) # normalize Unicode
      if unicodedata.category(c) != 'Mn' # cut non spacing marks
  )

def clean_string(string):
  string = string.lower().strip() # lowercase and get rid of excess white
  string = unicode_to_ASCII(string) # convert Unicode to ASCII
  string = re.sub(r"([.!?])", r" \1", string) # insert space before punct
  string = re.sub(r"[^a-zA-Z!?]+", r" ", string) # remove non alphabetic

  return string.strip()

def remove_punc(pairs):
  punctuation_pattern = r'[^\w\s]' # regex punctuation patterns
  cleaned_list = []
```

```
  # Remove punctuation from pairs
  for sublist in pairs:
    cleaned_sublist = [re.sub(punctuation_pattern, '', word) for word in
    cleaned_list.append(cleaned_sublist)

  return cleaned_list
```

Reading The File

Need to strip the CC tags from the file and split English and German sentences into pairs.

In [6]:

```
def read_file_langs(lang_1, lang_2, switch=False):
  print("Reading file...")
  # Strip whitespace and spit into lines

  # File Options: Full set, Test set respectively
  lines = open('/content/drive/My Drive/Translation-RNN/deu.txt',
               encoding='utf-8').read().strip().split('\n')
  # lines = open('/content/drive/My Drive/Translation-RNN/deu_test_set.tx
  #               encoding='utf-8').read().strip().split('\n')

  # Remove the CC tag from each line
  clean_lines = []
  for line in lines:
    clean_lines.append(remove_CC(line))

  # Split the lines into English German pairs
  split_lines = []
  for line in clean_lines:
    split_lines.append(line.split('\t'))

  sentence_pairs = remove_punc(split_lines)

  # Switch case: allows for reverse translation
  if not switch:
    native_lang = Language(lang_1)
    output_lang = Language(lang_2)
  else:
    sentence_pairs = [list(reversed(pair)) for pair in sentence_pairs]
    native_lang = Language(lang_1)
    output_lang = Language(lang_2)

  return native_lang, output_lang, sentence_pairs

# This accounts for the 'CC-BY 2.0 (France) Attribution: tatoeba.org #287
# (CM) & #8597805 (Roujin)' found in the dataset
def remove_control_characters(text):
    # Remove control characters from the text using regular expression
    cleaned_text = re.sub(r'[\x00-\x1F\x7F-\x9F]', '', text)
    return cleaned_text
```

Training Data Streamlining

Going to break sentences down to 12 word maximum.

In [7]:

```
# Max Sentence Length
SENTENCE_LENGTH = 12
```

Remove CC

> This particular dataset has a third field in each line that needs to be removed prior to generating the pairs.
>
> Format...
>
> Go. Geh. CC-BY 2.0 (France) Attribution: tatoeba.org #2877272 (CM) & #8597805 (Roujin)

In [8]:
```
def remove_CC(line):
  cleaned_line = re.sub(r'^(.*?)\t(.*?)\t.*$', r'\1\t\2', line)

  return cleaned_line
```

Full Preprocessing

> Put everything together

In [9]:
```
def preprocess_data(lang_1, lang_2, switch=False):
  native_lang, output_lang, pairs = read_file_langs(lang_1, lang_2, switc
  print(f'Read {len(pairs)} sentence pairs')
  print(f'Trimmed to {len(pairs)} pairs of {SENTENCE_LENGTH} words long (
  # Add each sentence to the language
  for pair in pairs:
    native_lang.add_sentence(pair[0])
    output_lang.add_sentence(pair[1])
  # Print word counts for reference
  print("Counted words:")
  print(native_lang.name, native_lang.word_count)
  print(output_lang.name, output_lang.word_count)

  return native_lang, output_lang, pairs

native_lang, output_lang, pairs = preprocess_data('eng', 'ger', False)
print(random.choice(pairs))
```

```
Reading file...
Read 271774 sentence pairs
Trimmed to 271774 pairs of 12 words long (max)
Counted words:
eng 20195
ger 42452
['Tom is a coward', 'Tom ist ein Feigling']
```

Encoder

The Encoder Decoder Network is a crucial architecture for language translation. This is because just looking at each input and producing an output (ex. word by word) doesn't capture the nuance of translation. Some sentence pairs have different lengths or have different structure based on how verbs are used.

Example

ENG: Did Tom mind? **GER:** Hatte Tom etwas einzuwenden?

The function of the encoder is to take the input and generate a context vector to generate the hidden state that will be used by the decoder. The dropout parametr will allow us to limit the model's dependency on certain words. This will randomly drop part of the input to force the model to learn more nuanced patterns for translation. It is important to note that dropout should not be too large to esnure the model retains proper accuracy.

This video was helpful in understanding the encoder functionality.

link: https://youtu.be/jCrgzJlxTKg?si=pDccqewYrWZzFOas

In [10]:
```
class EncoderRNN(nn.Module):
  # Initialize as subclass of nn.Module
  def __init__(self, input_size, hidden_size, pr_dropout = 0.1):
    super(EncoderRNN, self).__init__()
    self.hidden_size = hidden_size
    # Initialize Encoder Layers
    self.embedding = nn.Embedding(input_size, hidden_size) # embedded
    self.gru = nn.GRU(hidden_size, hidden_size, batch_first = True) # GRU
    self.dropout = nn.Dropout(pr_dropout) # dropout

  def forward(self, input):
    # Create embedded layer and perform dropout
    embedded = self.embedding(input)
    embedded_dropout = self.dropout(embedded)
    # Pass into GRU and return output and hidden state
    output, hidden_state = self.gru(embedded_dropout)

    return output, hidden_state
```

Decoder

The Encoder Decoder Network is a crucial architecture for language translation. This is because just looking at each input and producing an output (ex. word by word) doesn't capture the nuance of translation. Some sentence pairs have different lengths or have different structure based on how verbs are used.

Example

ENG: Did Tom mind? **GER:** Hatte Tom etwas einzuwenden?

The decoder takes the previous hidden state as well as the input and feeds these into the GRU (with the assistance of activation and normalization functions) to generate the output sequence in addition to an updated hidden state that will be used in the next forward step.

The YouTube channel that I used for the encoder has a good illustration of the decoder as well.

In [11]:

```
class DecoderRNN(nn.Module):
  # Initialize as subclass of nn.Module
  def __init__(self, hidden_size, output_size):
    super(DecoderRNN, self).__init__()
    # Initialize Decoder Layers
    self.embedding = nn.Embedding(output_size, hidden_size)
    self.gru = nn.GRU(hidden_size, hidden_size, batch_first = True)
    self.out = nn.Linear(hidden_size, output_size)

  def forward(self, encoder_outputs, encoder_hidden, target_tensor=None):
    # Encoder Outputs: (batch_size, sequence_length, hidden_size)
    batch_size = encoder_outputs.size(0)
    # Create input vector of SOS tokens
    decoder_in = torch.empty(batch_size, 1, dtype = torch.long, device =
    # Transfer hidden state
    decoder_hidden = encoder_hidden
    decoder_out = []
    # Step through sentence and append decoder outputs
    for idx in range(SENTENCE_LENGTH):
      decoder_output, decoder_hidden  = self.forward_step(decoder_in, dec
      decoder_out.append(decoder_output)
      # Teacher forcing
      if target_tensor is not None:
        decoder_in = target_tensor[:, idx].unsqueeze(1)
      # Use decoder output for new decoder input
      else:
        best_out = (decoder_output.topk(1))[1] # best predicted output
        decoder_in = best_out.squeeze(-1).detach() # feed into input
    # Concat decoder outputs and apply softmax
    decoder_out = torch.cat(decoder_out, dim = 1) # sequence dim
    norm_decoder_out = F.log_softmax(decoder_out, dim = -1)

    return norm_decoder_out, decoder_hidden, None

  # Each substep of a forward pass
  def forward_step(self, input, hidden):
    output = self.embedding(input) # embed input
    output = F.relu(output) # activation function
    output, hidden_state = self.gru(output, hidden) # feed into GRU
    output = self.out(output) # get decoder output

    return output, hidden_state
```

Attention

We are using Luong Attention. Luong Attention and Bahdanau Attention are the most common with the main difference being that Luong uses the current and past hidden states whareas Bahdnau only uses the past hidden state. Below is a link for a more in depth comparison of the two.

Link - https://stackoverflow.com/questions/44238154/what-is-the-difference-between-luong-attention-and-bahdanau-attention

A very brief description of attention is that it is a method that essentially allows the model to focus on certain parts of the sequence. This is acheived by calculating attention weights and applying them to the encoder output.

In [12]:
```
class LuongAttention(nn.Module):
  # Initialize as subclass of nn.Module
  def __init__(self, hidden_size):
    super(LuongAttention, self).__init__()
    self.Wa = nn.Linear(hidden_size, hidden_size)

  # Q = Query, K = Keys
  def forward(self, Q, K):
    Q_transformed = self.Wa(Q)

    # Test to verify size
    # print("Q size:", Q_transformed.size())
    # print("K size:", K.T.size())

    # Last dim is seq length
    attn_scores = torch.matmul(Q, K.transpose(1, 2))
    attn_weights = F.softmax(attn_scores, dim = -1)
    # Batch matric mult
    context = torch.bmm(attn_weights, K)

    return context, attn_weights
```

In [13]:
```
class AttnDecoderRNN(nn.Module):
  # Initialize as subclass of nn.Module
  def __init__(self, hidden_size, output_size, pr_dropout = 0.1):
    super(AttnDecoderRNN, self).__init__()
    self.embedding = nn.Embedding(output_size, hidden_size) #embedded lay
    # self.attention = BahdanauAttention(hidden_size)
    self.attention = LuongAttention(hidden_size) # attn mechanism
    self.gru = nn.GRU(2 * hidden_size, hidden_size, batch_first = True) #
    self.out = nn.Linear(hidden_size, output_size) # output layer
    self.dropout = nn.Dropout(pr_dropout) # perform dropout

  def forward(self, encoder_outputs, encoder_hidden, target_tensor = None
    batch_size = encoder_outputs.size(0) # get batch size
    # Input vector of SOS tokens
    decoder_input = torch.empty(batch_size, 1, dtype=torch.long,
                                device = device).fill_(SOS_Token)
    # Pass (t - 1) hidden into (t) hidden
    decoder_hidden = encoder_hidden
    decoder_out, attentions = [], []

    # Step through sentence and append decoder outputs
    for i in range(SENTENCE_LENGTH):
      # Perform forward step
      decoder_output, decoder_hidden, attn_weights = self.forward_step(
          decoder_input, decoder_hidden, encoder_outputs)

      decoder_out.append(decoder_output) # append dec output for idx
      attentions.append(attn_weights) # append attn_weights fro idx

      # Teacher forcing
      if target_tensor is not None:
        decoder_input = target_tensor[:, i].unsqueeze(1)
      # Use decoder output for new decoder input
      else:
        best_out = (decoder_output.topk(1))[1]
        decoder_input = best_out.squeeze(-1).detach()

    # Concat outputs along 2nd dim
    decoder_out = torch.cat(decoder_out, dim = 1)
```

```
    # Apply softmax (normalize)
    norm_decoder_out = F.log_softmax(decoder_out, dim = -1)
    # Concat attention weights along 2nd dim
    attentions = torch.cat(attentions, dim = 1)

    return norm_decoder_out, decoder_hidden, attentions

    # Each substep of a forward pass
  def forward_step(self, input, hidden, encoder_outputs):
    # Create embedded and perform dropout
    embedded = self.embedding(input)
    dropout_embedded = self.dropout(embedded)
    # Obtain context vec and attn_weights
    Q = hidden.permute(1, 0, 2)
    context, attn_weights = self.attention(Q, encoder_outputs)
    # Concat context to input for GRU input
    gru_in = torch.cat((embedded, context), dim = 2)
    # Run through GRU and output layer (linear)
    output, hidden = self.gru(gru_in, hidden)
    output = self.out(output)

    return output, hidden, attn_weights
```

Training Setup

Here we are essesntially just obtaining the indices for the input and target words using the structures defined in the preproccessing portion.

In [14]:

```
# Get words' corresponding indices
def indexesFromSentence(lang, sentence):
  idcs = []
  for word in sentence.split(' '):
    idcs.append(lang.word_to_idx[word])

  return idcs

# Create tensor from indices
def tensorFromSentence(lang, sentence):
  idcs = indexesFromSentence(lang, sentence)
  idcs.append(EOS_Token) # append EOS
  # Infer number of columns (since sentence size is variable)
  tens = torch.tensor(idcs, dtype = torch.long, device = device).view(1,

  return tens

# Create tensor pair from sentence pair
def tensorsFromPair(pair):
  input_tensor = tensorFromSentence(input_lang, pair[0])
  target_tensor = tensorFromSentence(output_lang, pair[1])

  return (input_tensor, target_tensor)

def get_dataloader(batch_size):
  # Preproccess
  input_lang, output_lang, pairs = preprocess_data('English', 'German', F
  # Initialize word indices
  n = len(pairs)
  input_ids = np.zeros((n, SENTENCE_LENGTH), dtype=np.int32)
```

```
    target_ids = np.zeros((n, SENTENCE_LENGTH), dtype=np.int32)
    # Create word indices
    for idx, (inp, tgt) in enumerate(pairs):
      inp_ids = indexesFromSentence(input_lang, inp)[:SENTENCE_LENGTH - 1]
      tgt_ids = indexesFromSentence(output_lang, tgt)[:SENTENCE_LENGTH - 1]
      inp_ids.append(EOS_Token) # append EOS token
      tgt_ids.append(EOS_Token) # append EOS token
      input_ids[idx, :len(inp_ids)] = inp_ids
      target_ids[idx, :len(tgt_ids)] = tgt_ids
    # Create training data
    training_data = TensorDataset(torch.LongTensor(input_ids).to(device),
                                  torch.LongTensor(target_ids).to(device))
    train_sampler = RandomSampler(training_data) # sample data randomly
    train_dataloader = DataLoader(training_data,
                                   sampler = train_sampler,
                                   batch_size = batch_size)

    return input_lang, output_lang, train_dataloader
```

Training

Now we continually feed snetence pairs to the model from the dataset. The time function simply aids with visualizing the progress of the model. It's important to note that out dataset is very large and you do not necessarily need to complete the entirety of the training to generate an accurate model. Additionally, you cn adjust the number of epochs/size of the dataset based on the accuracy you desire. Using the GPU is almost necessary for this step or the training could take hours.

You can play aorund with certain (non learnable) model paramaters such as dropout rate. Or you can altogether use a different model architecture (such as LSTM).

In [15]:
```
def train_epoch(dataloader, encoder, decoder, encoder_optimizer,
          decoder_optimizer, criterion):
    total_loss = 0
    # Iterate over batches
    for data in dataloader:
      input_tensor, target_tensor = data
      # Zero gradients
      encoder_optimizer.zero_grad()
      decoder_optimizer.zero_grad()
      # Pass through encoder and decoder
      encoder_outputs, encoder_hidden = encoder(input_tensor)
      decoder_outputs = decoder(encoder_outputs, encoder_hidden, target_ten
      loss = criterion(
        decoder_outputs.view(-1, decoder_outputs.size(-1)),
        target_tensor.view(-1)
      )
      loss.backward()
      # Optimize the gradients
      encoder_optimizer.step()
      decoder_optimizer.step()
      # Accumulate loss
      total_loss += loss.item()

    return total_loss / len(dataloader)
```

In [16]:

```python
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))
```

In [17]:

```python
def train(train_dataloader, encoder, decoder, n_epochs, learning_rate=0.0
               print_every=25, plot_every=100):
  start = time.time()
  plot_losses = []
  print_loss_total = 0
  plot_loss_total = 0
  # Initialize stochastic optimizer
  encoder_optimizer = optim.Adam(encoder.parameters(), lr = learning_rate
  decoder_optimizer = optim.Adam(decoder.parameters(), lr = learning_rate
  criterion = nn.NLLLoss()
  # Iterate over epochs
  for epoch in range(1, n_epochs + 1):
    loss = train_epoch(train_dataloader, encoder, decoder,
                       encoder_optimizer, decoder_optimizer, criterion)
    print_loss_total += loss
    plot_loss_total += loss
    # Print and plot loss over designated interval
    if epoch % print_every == 0:
      print_loss_avg = print_loss_total / print_every
      print_loss_total = 0
      print('%s (%d %d%%) %.3f' % (timeSince(start, epoch / n_epochs),
                                   epoch, epoch / n_epochs * 100, print_
    if epoch % plot_every == 0:
      plot_loss_avg = plot_loss_total / plot_every
      plot_losses.append(plot_loss_avg)
      plot_loss_total = 0

  showPlot(plot_losses)
```

In [18]:

```python
import matplotlib.pyplot as plt
plt.switch_backend('agg')
import matplotlib.ticker as ticker
import numpy as np

# Plot the loss over the epochs
def showPlot(points):
  plt.figure()
  fig, ax = plt.subplots()
  loc = ticker.MultipleLocator(base=0.2)
  ax.yaxis.set_major_locator(loc)
  plt.plot(points)
```

Evaluation

Essentially the training process but without the use of target vectors (basically the model has no feedback assistance now).

In [19]:
```
def evaluate(encoder, decoder, sentence, input_lang, output_lang):
  with torch.no_grad():
    input_tensor = tensorFromSentence(input_lang, sentence)
    # Pass through encoder and decoder
    encoder_outputs, encoder_hidden = encoder(input_tensor)
    decoder_outputs, decoder_hidden, decoder_attn = decoder(encoder_outpu
    # Get best k
    topi = decoder_outputs.topk(1)[1]
    decoded_ids = topi.squeeze()
    # Append word and EOS token from indices
    decoded_words = []
    for idx in decoded_ids:
      if idx.item() == EOS_Token:
        decoded_words.append('<EOS>')
        break # break out when hit end of sentence
      decoded_words.append(output_lang.idx_to_word[idx.item()])
  return decoded_words, decoder_attn
```

In [20]:
```
def evaluateRandomly(encoder, decoder, n = 10):
  # Evaluate the batches randomly
  for i in range(n):
    pair = random.choice(pairs)
    print('Input sequence:', pair[0])
    print('Correct translation:', pair[1])
    output_words = evaluate(encoder, decoder, pair[0], input_lang, output
    output_sentence = ' '.join(output_words)
    print('Generated translation', output_sentence)
    print('')
```

In [21]:
```
import torchvision
import torchaudio
# Set hidden size and batch size
hidden_size = 128
batch_size = 32
# Use GPU if available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
# Load data
input_lang, output_lang, train_dataloader = get_dataloader(batch_size)
# Initialize encoder and decoder
encoder = EncoderRNN(input_lang.word_count, hidden_size).to(device)
decoder = AttnDecoderRNN(hidden_size, output_lang.word_count).to(device)
# To ensure we are training on GPU
print("Training on:", device)
# Train
train(train_dataloader, encoder, decoder, 10, print_every = 1, plot_every
```

```
Reading file...
Read 271774 sentence pairs
Trimmed to 271774 pairs of 12 words long (max)
Counted words:
English 20195
German 42452
Training on: cuda
3m 12s (- 28m 51s) (1 10%) 2.392
6m 23s (- 25m 33s) (2 20%) 1.653
9m 34s (- 22m 19s) (3 30%) 1.391
12m 44s (- 19m 7s) (4 40%) 1.235
15m 56s (- 15m 56s) (5 50%) 1.130
19m 7s (- 12m 44s) (6 60%) 1.049
22m 17s (- 9m 33s) (7 70%) 0.986
25m 28s (- 6m 22s) (8 80%) 0.934
28m 40s (- 3m 11s) (9 90%) 0.890
31m 50s (- 0m 0s) (10 100%) 0.852
```

In [22]:

```
encoder.eval()
decoder.eval()
evaluateRandomly(encoder, decoder)
```

```
Input sequence: Tom mustve thought Mary didnt need to do that
Correct translation: Tom hat bestimmt gedacht Maria müsse das nicht mache
n
Generated translation Tom hat Maria gesagt dass er das nicht tun müsse <E
OS>

Input sequence: When did you last hear from Tom
Correct translation: Wann haben Sie das letzte Mal etwas von Tom gehört
Generated translation Wann hast du gestern von Tom gehört <EOS>

Input sequence: Tom is in the other room unpacking boxes
Correct translation: Tom ist im Zimmer nebenan und packt Kisten aus
Generated translation Tom ist in der Zimmer nur halb drei Kisten <EOS>

Input sequence: He knows how to milk a cow
Correct translation: Er weiß wie man eine Kuh melkt
Generated translation Er weiß wie man einen kleinen Fehler macht <EOS>

Input sequence: Stand up
Correct translation: Stehen Sie auf
Generated translation Steht auf links auf deinen Bett <EOS>

Input sequence: He was stunned by her beauty
Correct translation: Er war überwältigt von ihrer Schönheit
Generated translation Er wurde von ihrer Schönheit Schönheit von ihrer Sc
hönheit unterhalten <EOS>

Input sequence: I knew Tom would do something stupid
Correct translation: Ich wusste dass Tom etwas Dummes machen würde
Generated translation Ich wusste dass Tom etwas Dummes würde machen würde
<EOS>

Input sequence: Tom had no idea how tired Mary was
Correct translation: Tom hatte keine Ahnung wie erschöpft Mary war
Generated translation Tom hatte keine Ahnung wie lange Maria war <EOS>

Input sequence: Where is your homework
Correct translation: Wo sind deine Hausaufgaben
Generated translation Wo ist deine Hausaufgaben in deine Hausaufgaben <EO
S>

Input sequence: My bike has been stolen
Correct translation: Mein Rad wurde gestohlen
Generated translation Mein Fahrrad wurde gestohlen zu haben <EOS>
```