Introduction

In this notebook, you will find code scaffolding for the seq2seq part of Homework 3 (code for the trees section of the assignment is released in another notebook). There are certain parts of the scaffolding marked with `# Your code here` comments where you can fill in code to perform the specified tasks. After implementing the methods in this notebook, you will need to design and perform experiments to evaluate each method and respond to the questions in the Homework 3 handout (available on Canvas). You should be able to complete this assignment without changing any of the scaffolding code, just writing code to fill in the scaffolding and run experiments.

Set up dependencies and data

Let's use google drive to save our trained models to (so that we don't have to retrain them seventeen times).

```
from google.colab import drive
drive.mount("/content/drive")

MODEL_FOLDER = "/content/drive/My Drive/mit-6864/hw3"
!mkdir -p "/content/drive/My Drive/mit-6864/hw3"
```

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

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mou
```

```
%%bash
git clone https://github.com/mit-6864/hw3.git
mkdir -p /content/hw3/data

pip install sacrebleu
```

```
Collecting sacrebleu
  Downloading https://files.pythonhosted.org/packages/7e/57/0c7ca4e31a126189dab99c19951
Collecting portalocker==2.0.0
  Downloading https://files.pythonhosted.org/packages/89/a6/3814b7107e0788040870e8825ee
Installing collected packages: portalocker, sacrebleu
Successfully installed portalocker-2.0.0 sacrebleu-1.5.1
Cloning into 'hw3'...
```

```
import sys
sys.path.append("/content/hw3")

import lab_utils

import torch
import numpy as np

device = "cuda" if torch.cuda.is_available() else "cpu"
assert device == "cuda"   # use gpu whenever you can!

seed = 42
np.random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
```

Part 1: Sequence-to-Sequence Model

In this lab, we will explore RNN-based sequence-to-sequence (seq2seq) models to perform machine translation (MT).

- **Task:** translate from Vietnamese to English
- **Model:** RNN-based encoder-decoder
- **Data:** Vietnamese-English dataset from IWSLT'15

Implementation Tasks:

1. Data Preprocessing (done by TAs)
2. **Encoder**
3. **Decoder**
4. EncoderDecoder (done by TAs)
5. Generator (done by TAs)
6. Training (done by TAs)
7. **Greedy Decoding**
8. Testing via BLEU (done by TAs)

Section 1: Data Preprocessing

No need to write any code in this section. But you are encouraged to read through this part to understand the data.

Download data

First, we download the dataset and put it in the `/content/hw3/data` folder.

```
# Download data
DATA_DIR = "/content/hw3/data"

!wget -nv -O "$DATA_DIR/train.en" https://nlp.stanford.edu/projects/nmt/data/iwslt15.en-vi/tr
!wget -nv -O "$DATA_DIR/train.vi" https://nlp.stanford.edu/projects/nmt/data/iwslt15.en-vi/tr
!wget -nv -O "$DATA_DIR/tst2013.en" https://nlp.stanford.edu/projects/nmt/data/iwslt15.en-vi/
!wget -nv -O "$DATA_DIR/tst2013.vi" https://nlp.stanford.edu/projects/nmt/data/iwslt15.en-vi/
!wget -nv -O "$DATA_DIR/vocab.en" https://nlp.stanford.edu/projects/nmt/data/iwslt15.en-vi/vc
!wget -nv -O "$DATA_DIR/vocab.vi" https://nlp.stanford.edu/projects/nmt/data/iwslt15.en-vi/vc
```

```
2021-04-08 13:34:45 URL:https://nlp.stanford.edu/projects/nmt/data/iwslt15.en-vi/train.
2021-04-08 13:34:48 URL:https://nlp.stanford.edu/projects/nmt/data/iwslt15.en-vi/train.
2021-04-08 13:34:48 URL:https://nlp.stanford.edu/projects/nmt/data/iwslt15.en-vi/tst201
2021-04-08 13:34:49 URL:https://nlp.stanford.edu/projects/nmt/data/iwslt15.en-vi/tst201
2021-04-08 13:34:50 URL:https://nlp.stanford.edu/projects/nmt/data/iwslt15.en-vi/vocab.
2021-04-08 13:34:50 URL:https://nlp.stanford.edu/projects/nmt/data/iwslt15.en-vi/vocab.
```

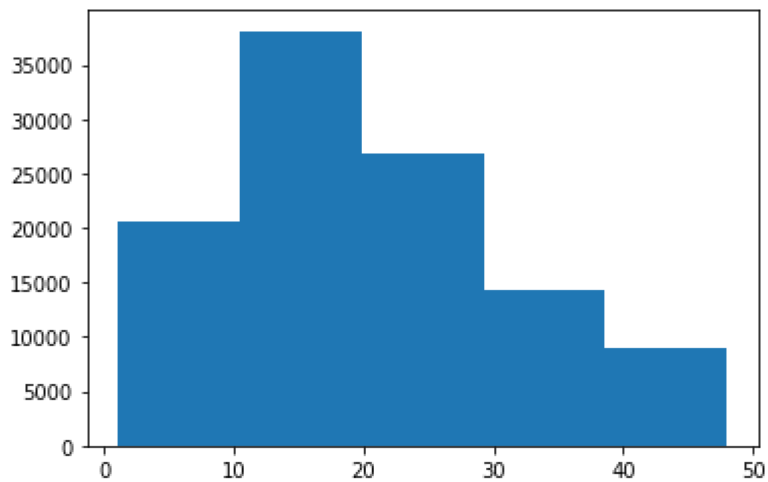
Load the Data and Preprocess

We then load the sentences and vocab lists, only keeping sentences that do not exceed 48 words (50 with the EOS tags).

```
from lab_utils import read_vocab_file, read_sentence_file, filter_data, show_some_data_stats

src_vocab_set = read_vocab_file("vocab.vi")
trg_vocab_set = read_vocab_file("vocab.en")

train_src_sentences_list = read_sentence_file("train.vi")
train_trg_sentences_list = read_sentence_file("train.en")
assert len(train_src_sentences_list) == len(train_trg_sentences_list)

test_src_sentences_list = read_sentence_file("tst2013.vi")
test_trg_sentences_list = read_sentence_file("tst2013.en")
assert len(test_src_sentences_list) == len(test_trg_sentences_list)

# Filter out sentences over 48 words long
MAX_SENT_LENGTH = 48
MAX_SENT_LENGTH_PLUS_SOS_EOS = 50

train_src_sentences_list, train_trg_sentences_list = filter_data(
    train_src_sentences_list, train_trg_sentences_list, MAX_SENT_LENGTH)
test_src_sentences_list, test_trg_sentences_list = filter_data(
```

```
test_src_sentences_list, test_trg_sentences_list = filter_data(
    test_src_sentences_list, test_trg_sentences_list, MAX_SENT_LENGTH)

# We take 10% of training data as validation set.
num_val = int(len(train_src_sentences_list) * 0.1)
val_src_sentences_list = train_src_sentences_list[:num_val]
val_trg_sentences_list = train_trg_sentences_list[:num_val]
train_src_sentences_list = train_src_sentences_list[num_val:]
train_trg_sentences_list = train_trg_sentences_list[num_val:]

show_some_data_stats(train_src_sentences_list, val_src_sentences_list,
                     test_src_sentences_list, train_trg_sentences_list,
                     src_vocab_set, trg_vocab_set)
```

```
Number of training (src, trg) sentence pairs: 108748
Number of validation (src, trg) sentence pairs: 12083
Number of testing (src, trg) sentence pairs: 1139
Size of en vocab set (including '<pad>', '<unk>', '<s>', '</s>'): 7711
Size of vi vocab set (including '<pad>', '<unk>', '<s>', '</s>'): 17193
Training sentence avg. length: 20
Training sentence length at 95-percentile: 42
Training sentence length distribution (x-axis is length range and y-axis is count):
```




```
Example Vietnamese input: ['Adam', 'Sadowsky', 'dàn', 'dựng', '1', 'video', 'âm', 'nhạc
Its target English output: ['Adam', 'Sadowsky', ':', 'How', 'to', 'engineer', 'a', 'vir
```

Define Dataset class

Here is the class for our dataset. We build off of the Dataset class. The IDs that we reserve might be useful later.

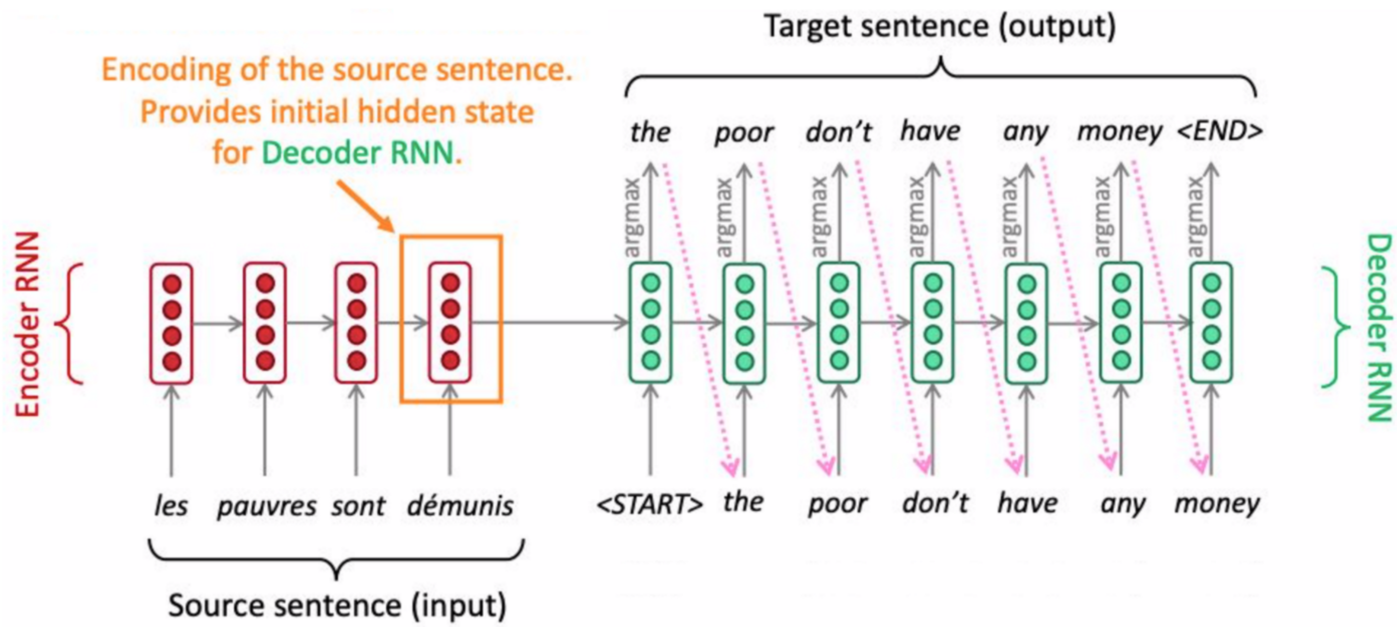
```
import torch
from torch.utils import data

# These IDs are reserved.
PAD_INDEX = 0
UNK_INDEX = 1
SOS_INDEX = 2
EOS_INDEX = 3
```

```
EOS_INDEX = 3


class MTDataset(data.Dataset):
  def __init__(self, src_sentences, src_vocabs, trg_sentences, trg_vocabs,
               sampling=1.):
    self.src_sentences = src_sentences[:int(len(src_sentences) * sampling)]
    self.trg_sentences = trg_sentences[:int(len(src_sentences) * sampling)]

    self.max_src_seq_length = MAX_SENT_LENGTH_PLUS_SOS_EOS
    self.max_trg_seq_length = MAX_SENT_LENGTH_PLUS_SOS_EOS

    self.src_vocabs = src_vocabs
    self.trg_vocabs = trg_vocabs

    self.src_v2id = {v : i for i, v in enumerate(src_vocabs)}
    self.src_id2v = {val : key for key, val in self.src_v2id.items()}  # the 1 is already res
    self.trg_v2id = {v : i for i, v in enumerate(trg_vocabs)}
    self.trg_id2v = {val : key for key, val in self.trg_v2id.items()}

  def __len__(self):
    return len(self.src_sentences)

  def __getitem__(self, index):
    src_sent = self.src_sentences[index]
    src_len = len(src_sent) + 2   # add <s> and </s> to each sentence
    src_id = []
    for w in src_sent:
      if w not in self.src_vocabs:
        w = '<unk>'
      src_id.append(self.src_v2id[w])
    src_id = ([SOS_INDEX] + src_id + [EOS_INDEX] + [PAD_INDEX] *
              (self.max_src_seq_length - src_len))

    trg_sent = self.trg_sentences[index]
    trg_len = len(trg_sent) + 2
    trg_id = []
    for w in trg_sent:
      if w not in self.trg_vocabs:
        w = '<unk>'
      trg_id.append(self.trg_v2id[w])
    trg_id = ([SOS_INDEX] + trg_id + [EOS_INDEX] + [PAD_INDEX] *
              (self.max_trg_seq_length - trg_len))

    return torch.tensor(src_id), src_len, torch.tensor(trg_id), trg_len
```

Section 2: Encoder

First, for a high-level overview. Our seq2seq model will consist of an Encoder RNN and a Decoder RNN. We will first implement this with no attention mechanism between the encoder and decoder. The encoder aims to compress the information contained in the entire input sequence into a single vector and pass it to the decoder.

Here's a picture overview if you're a visual person.



First let's implement the encoder, which in our case is just an RNN (feel free to use a GRU or try other cell types! and feel free to experiment with number of layers).

```
import torch.nn as nn
import torch.nn.functional as F
from torch.nn.utils.rnn import pack_padded_sequence, pad_packed_sequence


class Encoder(nn.Module):
  def __init__(self, input_size, hidden_size, dropout=0.):
    """
    Inputs:
      - `input_size`: an int representing the RNN input size.
      - `hidden_size`: an int representing the RNN hidden size.
      - `dropout`: a float representing the dropout rate during training. Note
          that for 1-layer RNN this has no effect since dropout only applies to
          outputs of intermediate layers.
    """

    super(Encoder, self).__init__()
    self.input_size = input_size
    self.hidden_size = hidden_size
    self.bidirectional = True
```

```
        self.bidirectional = True
        self.num_layers=2
        self.rnn = torch.nn.GRU(input_size = input_size, hidden_size=hidden_size, batch_first=Tru
        self.directions = 2 if self.bidirectional else 1


    def forward(self, inputs, lengths):
        """
        Inputs:
          - `inputs`: a 3d-tensor of shape (batch_size, max_seq_length, embed_size)
              representing a batch of padded embedded word vectors of source
              sentences.
          - `lengths`: a 1d-tensor of shape (batch_size,) representing the sequence
              lengths of `inputs`.

        Returns:
          - `outputs`: a 3d-tensor of shape
            (batch_size, max_seq_length, hidden_size).
          - `finals`: a 3d-tensor of shape (num_layers, batch_size, hidden_size).

          Hint: `outputs` and `finals` are both standard GRU outputs.
        """
        outputs = None
        finals = None

        # --------- Your code here --------- #
        # hint: you probably want to pack the inputs and outputs (see note below)
        #       https://pytorch.org/docs/stable/generated/torch.nn.utils.rnn.pack_padded_sequence
        # hint2: given the shape of the inputs and outputs,
        #        it might be helpful to specify batch_first=True (also in __init___)
        # hint3: MAX_SENT_LENGTH_PLUS_SOS_EOS is a global variable that exists if
        #        you ever need to specify a total_length for outputs

        padded_sequence = torch.nn.utils.rnn.pack_padded_sequence(inputs, lengths.to('cpu'), batc
        outputs, finals = self.rnn(padded_sequence)  # the initial hidden state is set to zero.

        # --------- Your code ends --------- #
        finals = torch.cat((finals[self.num_layers:, :, :], finals[:self.num_layers, :, :]), -1)
        return outputs, finals
```

Note about packing & padding:

Why we pad: to be able to batch sequences of different lengths

Why we pack: to be able do computations with padded sequences more efficiently

The second answer on this stackoverflow article is very helpful.

Section 3: Decoder

Here you will implement a decoder RNN.

At every step of decoding, the decoder is given an input token and hidden state. The initial input token is the start-of-string `<SOS>` token, and the first hidden state is the context vector (the encoder's last hidden state).

```
class Decoder(nn.Module):
  """An RNN decoder without attention."""

  def __init__(self, input_size, hidden_size, dropout=0.):
    """
      Inputs:
        - `input_size`, `hidden_size`, and `dropout` the same as in Encoder.
    """
    super(Decoder, self).__init__()

    # --------- Your code here --------- #
    # hint: you need more layers than the encoder
    #       again, feel free to use pytorch implemetnations
    #       https://pytorch.org/docs/stable/generated/torch.nn.GRU.html

    # To initialize from the final encoder state.
    self.input_size = input_size
    self.hidden_size = hidden_size
    self.num_layers = 2
    self.bridge = torch.nn.Linear(in_features=2*self.hidden_size, out_features=self.hidden_si
    #self.bridge = torch.nn.Linear(in_features=self.hidden_size, out_features=self.hidden_siz
    self.rnn = torch.nn.GRU(input_size=input_size, hidden_size=hidden_size, batch_first=True,
    # --------- Your code ends --------- #

  def forward_step(self, prev_embed, hidden):
    """Helper function for forward below:
       Perform a single decoder step (1 word).

       Inputs:
      - `prev_embed`: a 3d-tensor of shape (batch_size, 1, embed_size)
          representing the padded embedded word vectors at this step in training
      - `hidden`: a 3d-tensor of shape (1, batch_size, hidden_size) representing
          the current hidden state.

      Returns:
      - `hidden`: a 3d-tensor of shape (1, batch_size, hidden_size)
          representing the current decoder hidden state.
      - `pre_output`: a 3d-tensor of shape (batch_size, 1, hidden_size)
          representing the total decoder output for one step
    """
    pre_output = None
    # --------- Your code here --------- #
    #print('prev_embed', prev_embed.shape)
    #print('Input hidden', hidden.shape)
    pre_output, hidden = self.rnn(prev_embed, hidden)
```

```
    #print('output step', pre_output.shape)
    #print('Output hidden', hidden.shape)
    # --------- Your code ends --------- #
    return pre_output, hidden

  def forward(self, inputs, encoder_finals, hidden=None, max_len=None):
    """Unroll the decoder one step at a time.

    Inputs:
      - `inputs`: a 3d-tensor of shape (batch_size, max_seq_length, embed_size)
          representing a batch of padded embedded word vectors of target
          sentences (for teacher-forcing during training).
      - `encoder_finals`: a 3d-tensor of shape
          (num_enc_layers, batch_size, hidden_size) representing the final
          encoder hidden states used to initialize the initial decoder hidden
          states.
      - `hidden`: a 3d-tensor of shape (1, batch_size, hidden_size) representing
          the value to be used to initialize the initial decoder hidden states.
          If None, then use `encoder_finals`.
      - `max_len`: an int representing the maximum decoding length.

    Returns:
      - `outputs`: a 3d-tensor of shape
          (batch_size, max_seq_length, hidden_size) representing the raw
          decoder outputs (before converting to a `trg_vocab_size`-dim vector).
          We will convert it later in a `Generator` below.
      - `hidden`: a 3d-tensor of shape (1, batch_size, hidden_size)
          representing the last decoder hidden state.
    """

    # The maximum number of steps to unroll the RNN.
    if max_len is None:
      max_len = inputs.size(1)

    # Initialize decoder hidden state.
    if hidden is None:
      hidden = self.init_hidden(encoder_finals)
    #print('Hidden size ', self.hidden_size)
    #print('Encoder finals before the bridge', encoder_finals.shape)
    #print('Init with bridge', hidden.shape)

    # --------- Your code here --------- #

    # Unroll the decoder RNN for `max_len` steps.
    # hint: use the above helper function forward_step that
    #       performs a single decoder step (1 word).
    outputs = torch.zeros(inputs.size(0), max_len, self.hidden_size)
    for step in range(max_len):
      input = inputs[:, step, :]
      input = input[:, None, :]
      output, hidden = self.forward_step(input, hidden)
      outputs[:, step, :] = output[:, 0, :]
```

```
      outputs[:, step, :] = output[:, 0, :]
      """
      Decoder output shape torch.Size([128, 49, 256])
    Decoder output hidden state shape torch.Size([1, 128, 256])
      """
    # --------- Your code ends --------- #
    #print('Decoder output shape', outputs.shape)
    #print('Decoder output hidden state shape', hidden.shape)
    #return outputs, hidden # to check if the ordering is consistent
    return hidden, outputs

  def init_hidden(self, encoder_finals):
    """Use encoder final hidden state to initialize decoder's first hidden
    state."""
    decoder_init_hiddens = torch.tanh(self.bridge(encoder_finals))

    return decoder_init_hiddens
```

We have defined a high level encoder-decoder class to wrap up sub-models, including encoder, decoder, generator, and src/trg embeddings.

You don't need to write code here, but please try to understand what is going on!

```
class EncoderDecoder(nn.Module):
  """A standard Encoder-Decoder architecture without attention.
  """
  def __init__(self, encoder, decoder, src_embed, trg_embed, generator):
    """
    Inputs:
      - `encoder`: an `Encoder` object.
      - `decoder`: a `Decoder` object.
      - `src_embed`: an nn.Embedding object representing the lookup table for
          input (source) sentences.
      - `trg_embed`: an nn.Embedding object representing the lookup table for
          output (target) sentences.
      - `generator`: a `Generator` object. Essentially a linear mapping. See
          the next code cell.
    """
    super(EncoderDecoder, self).__init__()

    self.encoder = encoder
    self.decoder = decoder
    self.src_embed = src_embed
    self.trg_embed = trg_embed
    self.generator = generator

  def forward(self, src_ids, trg_ids, src_lengths):
    """Take in and process masked source and target sequences.

    Inputs:
      `src_ids`: a 2d-tensor of shape (batch_size, max_seq_length) representing
```

```
      `src_ids`: a 2d-tensor of shape (batch_size, max_seq_length) representing
        a batch of source sentences of word ids.
      `trg_ids`: a 2d-tensor of shape (batch_size, max_seq_length) representing
        a batch of target sentences of word ids.
      `src_lengths`: a 1d-tensor of shape (batch_size,) representing the
        sequence length of `src_ids`.

    Returns the decoder outputs, see the above cell.
    """
    encoder_hiddens, encoder_finals  = self.encode(src_ids, src_lengths)
    del encoder_hiddens   # unused
    return self.decode(encoder_finals, trg_ids[:, :-1])

  def encode(self, src_ids, src_lengths):
    return self.encoder(self.src_embed(src_ids), src_lengths)

  def decode(self, encoder_finals, trg_ids, decoder_hidden=None):
    return self.decoder(self.trg_embed(trg_ids), encoder_finals, decoder_hidden)
```

It simply projects the pre-output layer (x in the forward function below) to obtain the output layer, so that the final dimension is the target vocabulary size.

```
class Generator(nn.Module):
  """Define standard linear + softmax generation step."""
  def __init__(self, hidden_size, vocab_size):
    super(Generator, self).__init__()
    self.proj = nn.Linear(hidden_size, vocab_size, bias=False)

  def forward(self, x):
    return F.log_softmax(self.proj(x), dim=-1)
```

Wahoo! Now you have a working EncoderDecoder model! If you scroll down to the training section, you can train your model and try it out on the dataset. (Warning, it performs pretty miserably without Attention :'( )

Section 4: Training

We provide training and testing scripts here. You might need to adapt them to fit your model implementation.

Apply the dataloader to the MTDataset, which is defined in `lab_utils.py`. Dataloader provides a convenient way to iterate through the whole dataset.

```
from torch.utils import data
```

```
batch_size = 128

# You can try on a smaller training set by setting a smaller `sampling`.
train_set = MTDataset(train_src_sentences_list, src_vocab_set,
                      train_trg_sentences_list, trg_vocab_set, sampling=1.)
train_data_loader = data.DataLoader(train_set, batch_size=batch_size,
                                    num_workers=4, shuffle=True)

val_set = MTDataset(val_src_sentences_list, src_vocab_set,
                    val_trg_sentences_list, trg_vocab_set, sampling=1.)
val_data_loader = data.DataLoader(val_set, batch_size=batch_size, num_workers=4,
                                  shuffle=False)
```

```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:477: UserWarning:
  cpuset_checked))
```

The main functions for training, here we use perplexity to evaluate the performance of the model. Although we provide the training scripts here, we strongly encoureage you to go through and understand the procedure.

```
import math


class SimpleLossCompute:
  """A simple loss compute and train function."""

  def __init__(self, generator, criterion, opt=None):
    self.generator = generator
    self.criterion = criterion
    self.opt = opt

  def __call__(self, x, y, norm):
    #print('Before Final Linear Layer and Softmax', x.shape)
    x = self.generator(x.to(device))
    #print('Before After Linear Layer and Softmax', x.shape)
    #print('y objective shape', y.shape)
    loss = self.criterion(x.contiguous().view(-1, x.size(-1)),
                          y.contiguous().view(-1).to(device))
    loss = loss / norm

    if self.opt is not None:  # training mode
      loss.backward()
      self.opt.step()
      self.opt.zero_grad()

    return loss.data.item() * norm
```

```
def run_epoch(data_loader, model, loss_compute, print_every):
  """Standard Training and Logging Function"""

  total_tokens = 0
  total_loss = 0

  for i, (src_ids_BxT, src_lengths_B, trg_ids_BxL, trg_lengths_B) in enumerate(data_loader):
    # We define some notations here to help you understand the loaded tensor
    # shapes:
    #   `B`: batch size
    #   `T`: max sequence length of source sentences
    #   `L`: max sequence length of target sentences; due to our preprocessing
    #        in the beginning, `L` == `T` == 50
    # An example of `src_ids_BxT` (when B = 2):
    #   [[2, 4, 6, 7, ..., 4, 3, 0, 0, 0],
    #    [2, 8, 6, 5, ..., 9, 5, 4, 3, 0]]
    # The corresponding `src_lengths_B` would be [47, 49].
    # Note that SOS_INDEX == 2, EOS_INDEX == 3, and PAD_INDEX = 0.

    src_ids_BxT = src_ids_BxT.to(device)
    src_lengths_B = src_lengths_B.to(device)
    trg_ids_BxL = trg_ids_BxL.to(device)

    del trg_lengths_B   # unused
    #print('Length expected',  src_lengths_B)
    _, output = model(src_ids_BxT, trg_ids_BxL, src_lengths_B)

    loss = loss_compute(x=output, y=trg_ids_BxL[:, 1:],
                        norm=src_ids_BxT.size(0))
    total_loss += loss
    total_tokens += (trg_ids_BxL[:, 1:] != PAD_INDEX).data.sum().item()

    if model.training and i % print_every == 0:
      print("Epoch Step: %d Loss: %f" % (i, loss / src_ids_BxT.size(0)))

  return math.exp(total_loss / float(total_tokens))


def train(model, num_epochs, learning_rate, print_every):
  # Set `ignore_index` as PAD_INDEX so that pad tokens won't be included when
  # computing the loss.
  criterion = nn.NLLLoss(reduction="sum", ignore_index=PAD_INDEX)
  optim = torch.optim.Adam(model.parameters(), lr=learning_rate)

  # Keep track of dev ppl for each epoch.
  dev_ppls = []

  for epoch in range(num_epochs):
    print("Epoch", epoch)

    model.train()
    train_ppl = run_epoch(data_loader=train_data_loader, model=model,
```

```
                        loss_compute=SimpleLossCompute(model.generator,
                                                        criterion, optim),
                        print_every=print_every)

    model.eval()
    with torch.no_grad():
      dev_ppl = run_epoch(data_loader=val_data_loader, model=model,
                        loss_compute=SimpleLossCompute(model.generator,
                                                        criterion, None),
                        print_every=print_every)
      print("Validation perplexity: %f" % dev_ppl)
      dev_ppls.append(dev_ppl)

  return dev_ppls
```

The main function to perform training. First let's train the vanilla seq2seq model (fyi, it took ~10 minutes to go through 10 epochs using colab gpus; using default parameters, epoch 0 validation perplexity was 75ish and epoch 9 was 36ish).

Feel free to save the model more frequently (by adding a couple of lines in train() above) or change the path that it is saved at.

EncoderDecoder Training

```
# Hyperparameters for contructing the encoder-decoder model.
embed_size = 256 # Each word will be represented as a `embed_size`-dim vector, tuned over a g
hidden_size = 256  # GRU hidden size, tuned over a grid of 5 values
dropout = 0.2 # tuned dropout
lr = 8e-4


name_model = "pure_seq2seq_GRU_2_layers_bi_concat_dropout_true_02_ds_256_embed_256_hidden_siz
pure_seq2seq = EncoderDecoder(
  encoder=Encoder(embed_size, hidden_size, dropout=dropout),
  decoder=Decoder(embed_size, hidden_size, dropout=dropout),
  src_embed=nn.Embedding(len(src_vocab_set), embed_size),
  trg_embed=nn.Embedding(len(trg_vocab_set), embed_size),
  generator=Generator(hidden_size, len(trg_vocab_set))).to(device)
train_model = False
if train_model:
  # Start training. The returned `dev_ppls` is a list of dev perplexity for each
  # epoch.
  pure_dev_ppls = train(pure_seq2seq, num_epochs=10, learning_rate=lrs[i],
                      print_every=100)

  torch.save(pure_seq2seq.state_dict(), MODEL_FOLDER+"/" + name_model)

  # Plot perplexity
```

```
  # Plot perplexity
  lab_utils.plot_perplexity(pure_dev_ppls)
else:
  pure_seq2seq.load_state_dict(torch.load(MODEL_FOLDER+"/" + name_model))
```

Section 5: Decoding

Now that we have a trained model, the next task is to decode the model output. This is non-trivial. For the sake of simplicity, we'll go for the naive, greedy approach.

For greedy decoding, you will generate (or "decode") the target sentence by simply taking the argmax over the decoder output at each time step.

```
def greedy_decode(model, src_ids, src_lengths, max_len):
  """Greedily decode a sentence for EncoderDecoder. Make sure to chop off the
     EOS token!"""

  with torch.no_grad():
    _, encoder_finals = model.encode(src_ids, src_lengths)
    prev_y = torch.ones(1, 1).fill_(SOS_INDEX).type_as(src_ids)

  outputs = []
  hidden = model.decoder.init_hidden(encoder_finals)
  input = prev_y
  # --------- Your code here --------- #
  for _ in range(max_len):
    hidden, output = model.decode(encoder_finals, input.to(device), hidden)
    probabilities = model.generator(output.to(device))
    token_decoded = torch.argmax(probabilities)
    outputs.append(token_decoded.item())
    if token_decoded.item() in [3, 47]:
      break
    input = torch.ones(1, 1).fill_(token_decoded).type_as(src_ids)
  # --------- Your code ends --------- #

  return filter(lambda x: x not in [3, 47], outputs)  # . </>
```

Let's look at three examples for the EncoderDecoder model. Feel free to play around here, printing out more examples.

```
example_set = MTDataset(val_src_sentences_list, src_vocab_set,
                        val_trg_sentences_list, trg_vocab_set)
example_data_loader = data.DataLoader(example_set, batch_size=1, num_workers=1,
                                      shuffle=False)
```

```
print("EncoderDecoder Results:")
lab_utils.print_examples(pure_seq2seq, src_vocab_set, trg_vocab_set,
```

```
lab_utils.print_examples(pure_seq2seq, src_vocab_set, trg_vocab_set,
                         example_data_loader, greedy_decode, n=3)
```

```
EncoderDecoder Results:
Example #1
Src :  Khoa học đằng sau một tiêu đề về khí hậu
Trg :  Rachel <unk> : The science behind a climate headline
Pred:  Science is a <unk> mystery of the <unk>

Example #2
Src :  Tôi muốn cho các bạn biết về sự to lớn của những nỗ lực khoa học đã góp phần làm
Trg :  I &apos;d like to talk to you today about the scale of the scientific effort tha
Pred:  I want to show you the story of the research that you &apos;ve learned about how

Example #3
Src :  Có những dòng trông như thế này khi bàn về biến đổi khí hậu , và như thế này khi
Trg :  <unk> that look like this when they have to do with climate change , and headlin
Pred:  There are projections like this , and this is like the <unk> , and when you look
```

Section 6: Testing

Compute the BLEU score on the test set. BLEU score is a standard measure to evaluate the translation results. For further details, you can refer to this link. (The TAs' preliminary implementation of EncoderDecoder gets a BLEU score of around 6).

```
import sacrebleu
from tqdm import tqdm

def compute_BLEU(model, data_loader, decoder, trg_vocab_set):

  bleu_score = []

  model.eval()
  for src_ids, src_lengths, trg_ids, _ in tqdm(data_loader):
    result = decoder(model, src_ids.to(device), src_lengths.to(device),
                     max_len=MAX_SENT_LENGTH_PLUS_SOS_EOS)

    # remove <s>
    src_ids = src_ids[0, 1:]
    trg_ids = trg_ids[0, 1:]
    # remove </s> and <pad>
    src_ids = src_ids[:np.where(src_ids == EOS_INDEX)[0][0]]
    trg_ids = trg_ids[:np.where(trg_ids == EOS_INDEX)[0][0]]

    pred = " ".join(lab_utils.lookup_words(result, vocab=trg_vocab_set))
    targ = " ".join(lab_utils.lookup_words(trg_ids, vocab=trg_vocab_set))

    bleu_score.append(sacrebleu.raw_corpus_bleu([pred], [[targ]], .01).score)
```

```
  return bleu_score


test_set = MTDataset(test_src_sentences_list, src_vocab_set,
                     test_trg_sentences_list, trg_vocab_set, sampling=1.)
test_data_loader = data.DataLoader(test_set, batch_size=1, num_workers=4,
                                   shuffle=False)

print('BLEU score without Attention: %f' % (np.mean(compute_BLEU(pure_seq2seq,
                                            test_data_loader,
                                            greedy_decode, trg_vocab_set))))
```

```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:477: UserWarning:
  cpuset_checked))
100%|██████████| 1139/1139 [00:20<00:00, 55.54it/s]BLEU score without Attention: 5.7921
```

We have performed exensive experiments in order to tune our model:

- Bidirectional cells and different ways of combining hidden states (sum and concatenation)
- Several number of layers for the GRU encoders and decoders
- Varying the dropout
- Varying the embedding size
- Varying the hidden size
- Varying the learning rate

Now, we will use our best model (ie from the kind of CAVI that we've done here) and try to implement attention on top of it. We will once again monitor our improvements using perplexity (we will explain why later on). We will only try one strategy of Attention and leave the other implementations of Attention as a further work.

Encoder Decoder with Attention

```
import torch.nn as nn
import torch.nn.functional as F
from torch.nn.utils.rnn import pack_padded_sequence, pad_packed_sequence

"""Note: from https://github.com/IBM/pytorch-seq2seq/issues/141, we could also include maskir
"""A global attention layer, as introduced by Luong et al. (2015)"""

class Attention(nn.Module):
  """Encoder hiddens torch.Size([128, 50, 512])
     Decoder hiddens torch.Size([128, 49, 256])"""
  def __init__(self, hidden_size):
    super(Attention, self).__init__()
```

```
    self.score_weights = nn.Parameter(torch.Tensor(hidden_size, hidden_size))
    self.bridge = nn.Linear(2*hidden_size, hidden_size)   # 2 again in order to take into acc
    self.normalizer = nn.Softmax(dim=-1)
    self.gate = nn.Tanh()

  def forward(self, encoder_outputs, decoder_outputs):
    scores = torch.bmm(decoder_outputs@self.score_weights, torch.transpose(encoder_outputs, 1
    # print('----INSIDE ATTENTION------')
    # print('Scores', scores.shape)
    attn_weights = self.normalizer(scores)
    # print('Normalized scores', attn_weights.shape)
    context = torch.bmm(attn_weights, encoder_outputs)
    attn_hidden = self.gate(self.bridge(torch.cat((decoder_outputs, context), dim=-1)))
    # print('Output', attn_hidden.shape)
    return attn_hidden



class Encoder(nn.Module):
  def __init__(self, input_size, hidden_size, dropout=0.):
    super(Encoder, self).__init__()
    self.input_size = input_size
    self.hidden_size = hidden_size
    self.bidirectional = True
    self.num_layers=2
    self.rnn = torch.nn.GRU(input_size = input_size, hidden_size=hidden_size, batch_first=Tru
    self.directions = 2 if self.bidirectional else 1


  def forward(self, inputs, lengths):
    outputs = None
    finals = None
    # padded_sequence = torch.nn.utils.rnn.pack_padded_sequence(inputs, lengths.to('cpu'), ba
    outputs, finals = self.rnn(inputs)  # the initial hidden state is set to zero.
    finals = torch.cat((finals[self.num_layers:, :, :], finals[:self.num_layers, :, :]), -1)
    # print('-------ENCODER-------')
    # print('Output shape', outputs.shape)
    # print('Finals', finals.shape)
    return outputs, finals




class Decoder_Attention(nn.Module):
  """An RNN decoder with attention.
  How is the decoder with Attention different ?
  We need to do one step at a time in order to compute everything, so no need for forward ste
  Inspired from https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html#

  def __init__(self, input_size, hidden_size, attention=None, dropout=0.):
    super(Decoder_Attention, self).__init__()
    self.input_size = input_size
    self.hidden_size = hidden_size
```

```
    self.attention = attention
    self.bridge = nn.Linear(in_features=2*self.hidden_size, out_features=self.hidden_size)
    self.num_layers = 2
    self.rnn = torch.nn.GRU(input_size=input_size, hidden_size=hidden_size, batch_first=True,


  def forward(self, inputs, encoder_hiddens, encoder_finals, hidden=None, max_len=None):
    if max_len is None:
      max_len = inputs.size(1)
    if hidden is None:
      hidden = self.init_hidden(encoder_finals)
    # print('-------DECODER-----')
    # print('Inputs', inputs.shape)
    # print('Encoder hiddens', encoder_hiddens.shape)
    # print('Encoder finals', encoder_finals.shape)
    # print('Hidden', hidden.shape)
    decoder_hiddens, hidden = self.rnn(inputs, hidden)
    encoder_hiddens = self.init_hidden(encoder_hiddens)
    # print('------DECODER FOR ATTENTION-------')
    # print('Encoder hiddens', encoder_hiddens.shape)
    # print('Decoder hiddens', decoder_hiddens.shape)
    # print('------OUTPUT------')
    # print('Hidden', hidden.shape)
    outputs = self.attention(encoder_hiddens, decoder_hiddens)
    # print('Outputs', outputs.shape)
    return hidden, outputs

  def init_hidden(self, encoder_finals):
    decoder_init_hiddens = torch.tanh(self.bridge(encoder_finals))
    return decoder_init_hiddens


class EncoderDecoder_Attention(nn.Module):
  def __init__(self, encoder, decoder, src_embed, trg_embed, generator):
    super(EncoderDecoder_Attention, self).__init__()
    self.encoder = encoder
    self.decoder = decoder
    self.src_embed = src_embed
    self.trg_embed = trg_embed
    self.generator = generator

  def forward(self, src_ids, trg_ids, src_lengths):
    encoder_hiddens, encoder_finals  = self.encode(src_ids, src_lengths)
    return self.decode(encoder_finals, trg_ids[:, :-1].long(), encoder_hiddens)

  def encode(self, src_ids, src_lengths):
    return self.encoder(self.src_embed(src_ids.long()), src_lengths)

  def decode(self, encoder_finals, trg_ids, encoder_hiddens):
    return self.decoder(self.trg_embed(trg_ids.long()), encoder_hiddens, encoder_finals)
```

```
batch_size = 128
train_set = MTDataset(train_src_sentences_list, src_vocab_set,
                      train_trg_sentences_list, trg_vocab_set, sampling=1.)
train_data_loader = data.DataLoader(train_set, batch_size=batch_size,
                                    num_workers=4, shuffle=True)
val_set = MTDataset(val_src_sentences_list, src_vocab_set,
                    val_trg_sentences_list, trg_vocab_set, sampling=1.)
val_data_loader = data.DataLoader(val_set, batch_size=batch_size, num_workers=4,
                                  shuffle=False)

# Hyperparameters for contructing the encoder-decoder model.
embed_size = 256 # Each word will be represented as a `embed_size`-dim vector, tuned over a g
hidden_size = 256  # GRU hidden size, tuned over a grid of 5 values
dropout = 0.2 # tuned dropout
lr = 8e-4



name_model = "pure_seq2seq_attention.pt"
pure_seq2seq_attention =  EncoderDecoder_Attention(
  encoder=Encoder(embed_size, hidden_size, dropout=dropout),
  decoder=Decoder_Attention(embed_size, hidden_size,
                attention=Attention(hidden_size), dropout=dropout),
  src_embed=nn.Embedding(len(src_vocab_set), embed_size),
  trg_embed=nn.Embedding(len(trg_vocab_set), embed_size),
  generator=Generator(hidden_size, len(trg_vocab_set))).to(device)
train_model = False
if train_model:
  # Start training. The returned `dev_ppls` is a list of dev perplexity for each
  # epoch.
  pure_dev_ppls = train(pure_seq2seq_attention, num_epochs=10, learning_rate=lr,
                        print_every=100)

  torch.save(pure_seq2seq_attention.state_dict(), MODEL_FOLDER+"/" + name_model)

  # Plot perplexity
  lab_utils.plot_perplexity(pure_dev_ppls)
else:
  pure_seq2seq_attention.load_state_dict(torch.load(MODEL_FOLDER+"/" + name_model))
```

```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:477: UserWarning:
  cpuset_checked))
```

Encoder Decoder with Attention & Masking

Inspired from https://github.com/IBM/pytorch-seq2seq/issues/141, we can see that without Masking the performances are not that great. Perhaprs leaving our model train for much longer will improve the performances, but for now we do not have very good results. We will see later on what

```
import torch.nn as nn
import torch.nn.functional as F
from torch.nn.utils.rnn import pack_padded_sequence, pad_packed_sequence

"""Note: from https://github.com/IBM/pytorch-seq2seq/issues/141, we could also include maskir
"""A global attention layer, as introduced by Luong et al. (2015)"""

class Mask_Attention(nn.Module):
  """Encoder hiddens torch.Size([128, 50, 512])
     Decoder hiddens torch.Size([128, 49, 256])"""
  def __init__(self, hidden_size):
    super(Mask_Attention, self).__init__()
    self.score_weights = nn.Parameter(torch.Tensor(hidden_size, hidden_size))
    self.bridge = nn.Linear(2*hidden_size, hidden_size)   # 2 again in order to take into acc
    self.normalizer = nn.Softmax(dim=-1)
    self.gate = nn.Tanh()

  def forward(self, encoder_outputs, decoder_outputs, input_mask=None, target_mask=None):
    """Here, input_mask and target_mask are masks in the input and the target in order to loc
    """
    scores = torch.bmm(decoder_outputs@self.score_weights, torch.transpose(encoder_outputs, 1
    # print('----INSIDE ATTENTION------')
    # print('Scores', scores.shape)
    max_len = decoder_outputs.size(1)
    if input_mask is not None:
      attn_mask = input_mask.expand(-1, max_len, -1)
      scores = scores+torch.log(attn_mask) # will give some -inf values, that will be zeroed
    attn_weights = self.normalizer(scores)
    # print('Normalized scores', attn_weights.shape)
    context = torch.bmm(attn_weights, encoder_outputs)
    attn_hidden = self.gate(self.bridge(torch.cat((decoder_outputs, context), dim=-1)))
    # print('Output', attn_hidden.shape)
    return attn_hidden



class Encoder(nn.Module):
  def __init__(self, input_size, hidden_size, dropout=0.):
    super(Encoder, self).__init__()
    self.input_size = input_size
    self.hidden_size = hidden_size
    self.bidirectional = True
    self.num_layers=2
    self.rnn = torch.nn.GRU(input_size = input_size, hidden_size=hidden_size, batch_first=Tru
    self.directions = 2 if self.bidirectional else 1


  def forward(self, inputs, lengths):
    outputs = None
    finals = None
```

```
    finals = None
    # padded_sequence = torch.nn.utils.rnn.pack_padded_sequence(inputs, lengths.to('cpu'), ba
    outputs, finals = self.rnn(inputs)  # the initial hidden state is set to zero.
    finals = torch.cat((finals[self.num_layers:, :, :], finals[:self.num_layers, :, :]), -1)
    # print('-------ENCODER-------')
    # print('Output shape', outputs.shape)
    # print('Finals', finals.shape)
    return outputs, finals



class Decoder_MaskAttention(nn.Module):
  """An RNN decoder with attention.
  How is the decoder with Attention different ?
  We need to do one step at a time in order to compute everything, so no need for forward ste
  Inspired from https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html#

  def __init__(self, input_size, hidden_size, attention=None, dropout=0.):
    super(Decoder_MaskAttention, self).__init__()
    self.input_size = input_size
    self.hidden_size = hidden_size
    self.attention = attention
    self.bridge = nn.Linear(in_features=2*self.hidden_size, out_features=self.hidden_size)
    self.num_layers = 2
    self.rnn = torch.nn.GRU(input_size=input_size, hidden_size=hidden_size, batch_first=True,


  def forward(self, inputs, encoder_hiddens, encoder_finals, input_mask, target_mask, hidden=
    if max_len is None:
      max_len = inputs.size(1)
    if hidden is None:
      hidden = self.init_hidden(encoder_finals)
    # print('-------DECODER-----')
    # print('Inputs', inputs.shape)
    # print('Encoder hiddens', encoder_hiddens.shape)
    # print('Encoder finals', encoder_finals.shape)
    # print('Hidden', hidden.shape)
    decoder_hiddens, hidden = self.rnn(inputs, hidden)
    encoder_hiddens = self.init_hidden(encoder_hiddens)
    # print('------DECODER FOR ATTENTION-------')
    # print('Encoder hiddens', encoder_hiddens.shape)
    # print('Decoder hiddens', decoder_hiddens.shape)
    # print('------OUTPUT------')
    # print('Hidden', hidden.shape)
    outputs = self.attention(encoder_hiddens, decoder_hiddens, input_mask, target_mask)
    # print('Outputs', outputs.shape)
    return hidden, outputs

  def init_hidden(self, encoder_finals):
    decoder_init_hiddens = torch.tanh(self.bridge(encoder_finals))
    return decoder_init_hiddens
```

```
class EncoderDecoder_MaskAttention(nn.Module):
  def __init__(self, encoder, decoder, src_embed, trg_embed, generator):
    super(EncoderDecoder_MaskAttention, self).__init__()
    self.encoder = encoder
    self.decoder = decoder
    self.src_embed = src_embed
    self.trg_embed = trg_embed
    self.generator = generator

  def forward(self, src_ids, trg_ids, src_lengths):
    src_mask = torch.where(src_ids == PAD_INDEX, torch.zeros(src_ids.size()).to(device), torc
    src_mask.unsqueeze_(1)
    trg_mask = torch.where(trg_ids[:, :-1] == PAD_INDEX, torch.zeros(trg_ids[:, :-1].size()).
    encoder_hiddens, encoder_finals  = self.encode(src_ids, src_lengths)
    return self.decode(encoder_finals, trg_ids[:, :-1].long(), encoder_hiddens, src_mask, trg

  def encode(self, src_ids, src_lengths):
    return self.encoder(self.src_embed(src_ids.long()), src_lengths)

  def decode(self, encoder_finals, trg_ids, encoder_hiddens, src_mask=None, trg_mask=None, de
    return self.decoder(self.trg_embed(trg_ids.long()), encoder_hiddens, encoder_finals, src_


batch_size = 128
train_set = MTDataset(train_src_sentences_list, src_vocab_set,
                      train_trg_sentences_list, trg_vocab_set, sampling=1.)
train_data_loader = data.DataLoader(train_set, batch_size=batch_size,
                                    num_workers=4, shuffle=True)
val_set = MTDataset(val_src_sentences_list, src_vocab_set,
                    val_trg_sentences_list, trg_vocab_set, sampling=1.)
val_data_loader = data.DataLoader(val_set, batch_size=batch_size, num_workers=4,
                                  shuffle=False)

# Hyperparameters for contructing the encoder-decoder model.
embed_size = 256 # Each word will be represented as a `embed_size`-dim vector, tuned over a g
hidden_size = 256  # GRU hidden size, tuned over a grid of 5 values
dropout = 0.2 # tuned dropout
lr = 1e-3



name_model = "pure_seq2seq_Maskattention.pt"
pure_seq2seq_maskattention =  EncoderDecoder_MaskAttention(
  encoder=Encoder(embed_size, hidden_size, dropout=dropout),
  decoder=Decoder_MaskAttention(embed_size, hidden_size,
                  attention=Mask_Attention(hidden_size), dropout=dropout),
  src_embed=nn.Embedding(len(src_vocab_set), embed_size),
  trg_embed=nn.Embedding(len(trg_vocab_set), embed_size),
  generator=Generator(hidden_size, len(trg_vocab_set))).to(device)
train_model = False
if train_model:
  # Start training. The returned `dev_ppls` is a list of dev perplexity for each
```

```
  # epoch.
  pure_dev_ppls = train(pure_seq2seq_maskattention, num_epochs=10, learning_rate=lr,
                      print_every=100)

  torch.save(pure_seq2seq_maskattention.state_dict(), MODEL_FOLDER+"/" + name_model)

  # Plot perplexity
  lab_utils.plot_perplexity(pure_dev_ppls)
else:
  pure_seq2seq_maskattention.load_state_dict(torch.load(MODEL_FOLDER+"/" + name_model))

    /usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:477: UserWarning:
      cpuset_checked))
```

Alternative strategy for decoding

I will implement here two alernative strategies for decoding : Top k sampling and Top p sampling. One last way of fine-tuning the decoder would be to use Beam Search, but we leave it as future avenues for our model to get better. DO not forget to divide by the sum in order to have probabilities that sum up to 1

For now, using the biLSTM

```
# Hyperparameters for contructing the encoder-decoder model.
embed_size = 256 # Each word will be represented as a `embed_size`-dim vector, tuned over a g
hidden_size = 256  # GRU hidden size, tuned over a grid of 5 values
dropout = 0.2 # tuned dropout
lr = 8e-4

name_model = "pure_seq2seq_GRU_2_layers_bi_concat_dropout_true_02_ds_256_embed_256_hidden_siz
pure_seq2seq = EncoderDecoder(
  encoder=Encoder(embed_size, hidden_size, dropout=dropout),
  decoder=Decoder(embed_size, hidden_size, dropout=dropout),
  src_embed=nn.Embedding(len(src_vocab_set), embed_size),
  trg_embed=nn.Embedding(len(trg_vocab_set), embed_size),
  generator=Generator(hidden_size, len(trg_vocab_set))).to(device)
pure_seq2seq.load_state_dict(torch.load(MODEL_FOLDER+"/" + name_model))

    <All keys matched successfully>
```

```
test_set = MTDataset(test_src_sentences_list, src_vocab_set,
                     test_trg_sentences_list, trg_vocab_set, sampling=1.)
test_data_loader = data.DataLoader(test_set, batch_size=1, num_workers=4,
                                   shuffle=False)
```

```
print('BLEU score without Attention: %f' % (np.mean(compute_BLEU(pure_seq2seq,
                                            test_data_loader,
                                            greedy_decode, trg_vocab_set))))
```

```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:477: UserWarning:
  cpuset_checked))
100%|██████████| 1139/1139 [00:55<00:00, 20.45it/s]BLEU score without Attention: 1.2203
```

Decoding with other models

```
example_set = MTDataset(val_src_sentences_list, src_vocab_set,
                        val_trg_sentences_list, trg_vocab_set)
example_data_loader = data.DataLoader(example_set, batch_size=1, num_workers=1,
                                      shuffle=False)
```

```
print("EncoderDecoder Results:")
lab_utils.print_examples(pure_seq2seq, src_vocab_set, trg_vocab_set,
                         example_data_loader, greedy_decode, n=3)
```

```
EncoderDecoder Results:
Example #1
Src :  Khoa học đằng sau một tiêu đề về khí hậu
Trg :  Rachel <unk> : The science behind a climate headline
Pred:  tremendously FN discovery Twelve time thyself time Jenkins experts thyself scien

Example #2
Src :  Tôi muốn cho các bạn biết về sự to lớn của những nỗ lực khoa học đã góp phần làm
Trg :  I &apos;d like to talk to you today about the scale of the scientific effort tha
Pred:  Malaysia melts 20s napot ago Angeles &apos;am &apos;am Google &apos;am &apos;am

Example #3
Src :  Có những dòng trông như thế này khi bàn về biến đổi khí hậu , và như thế này khi
Trg :  <unk> that look like this when they have to do with climate change , and headlin
Pred:  Kenya flames Shake Google technologist technologist course course farmer farmer
```

```
# Hyperparameters for contructing the encoder-decoder model.
embed_size = 256 # Each word will be represented as a `embed_size`-dim vector, tuned over a g
hidden_size = 256  # GRU hidden size, tuned over a grid of 5 values
dropout = 0.2 # tuned dropout
lr = 8e-4

name_model = "pure_seq2seq_GRU_2_layers_bi_concat_dropout_true_02_ds_256_embed_256_hidden_siz
pure_seq2seq = EncoderDecoder(
  encoder=Encoder(embed_size, hidden_size, dropout=dropout),
  decoder=Decoder(embed_size, hidden_size, dropout=dropout),
```

```
      src_embed=nn.Embedding(len(src_vocab_set), embed_size),
      trg_embed=nn.Embedding(len(trg_vocab_set), embed_size),
      generator=Generator(hidden_size, len(trg_vocab_set))).to(device)
    pure_seq2seq.load_state_dict(torch.load(MODEL_FOLDER+"/" + name_model))


    def greedy_decode(model, src_ids, src_lengths, max_len):
      """Greedily decode a sentence for EncoderDecoder. Make sure to chop off the
         EOS token!"""

      with torch.no_grad():
        _, encoder_finals = model.encode(src_ids, src_lengths)
        prev_y = torch.ones(1, 1).fill_(SOS_INDEX).type_as(src_ids)

      outputs = []
      hidden = model.decoder.init_hidden(encoder_finals)
      input = prev_y
      # --------- Your code here --------- #
      for _ in range(max_len):
        hidden, output = model.decode(encoder_finals, input.to(device), hidden)
        probabilities = model.generator(output.to(device))
        token_decoded = torch.argmax(probabilities)
        outputs.append(token_decoded.item())
        if token_decoded.item() in [3, 47]:
          break
        input = torch.ones(1, 1).fill_(token_decoded).type_as(src_ids)
      # --------- Your code ends --------- #

      return filter(lambda x: x not in [3, 47], outputs)  # . </>


    def topk_decode(model, src_ids, src_lengths, max_len):
      with torch.no_grad():
        _, encoder_finals = model.encode(src_ids, src_lengths)
        prev_y = torch.ones(1, 1).fill_(SOS_INDEX).type_as(src_ids)
      k = 20
      outputs = []
      hidden = model.decoder.init_hidden(encoder_finals)
      input = prev_y
      for _ in range(max_len):
        hidden, output = model.decode(encoder_finals, input.to(device), hidden)
        probabilities = model.generator(output.to(device))
        topk_proba_indices = torch.topk(probabilities, k).indices
        top_k_proba = probabilities[topk_proba_indices]
        renormalized_proba = top_k_proba/top_k_proba.sum() # here, we have a distribution over th
        print('Renormalized proba', renormalized_proba.to('cpu'))
        selected_token = np.random.choice(a=topk_proba_indices.numpy(), size=1, p=renormalized_pr
        outputs.append(selected_token)
        if token_decoded.item() in [3, 47]:
          break
        input = torch.ones(1, 1).fill_(token_decoded).type_as(src_ids)
      return torch.tensor(list(filter(lambda x: x not in [3, 47], outputs)), device=device)  # .
```

```
def topp_decode(model, src_ids, src_lengths, max_len):
  with torch.no_grad():
    _, encoder_finals = model.encode(src_ids, src_lengths)
    prev_y = torch.ones(1, 1).fill_(SOS_INDEX).type_as(src_ids)
  outputs = []
  threshold = 0.8
  hidden = model.decoder.init_hidden(encoder_finals)
  input = prev_y
  for _ in range(max_len):
    hidden, output = model.decode(encoder_finals, input.to(device), hidden)
    probabilities = model.generator(output.to(device)).numpy()
    argsort_proba = torch.argsort(probabilities).numpy()
    sorted_proba = probabilities[argsort_prob].cpu().numpy()
    cumulated_proba = np.cumsum(sorted_proba)
    index_threshold = np.argmin(cumulated_proba > threshold)
    selected_indexes = argsort_proba[:index_threshold]
    selected_probabilities = probabilities[selected_indexes]
    distribution = selected_probabilities/np.sum(selected_probabilities)
    selected_token=np.random.choice(a=selected_indexes, size=1, p=distribution)
    outputs.append(selected_token)
    if token_decoded.item() in [3, 47]:
      break
    input = torch.ones(1, 1).fill_(token_decoded).type_as(src_ids)
  return filter(lambda x: x not in [3, 47], outputs)  # . </>
```

Reference

```
test_set = MTDataset(test_src_sentences_list, src_vocab_set,
                     test_trg_sentences_list, trg_vocab_set, sampling=1.)
test_data_loader = data.DataLoader(test_set, batch_size=1, num_workers=4,
                                   shuffle=False)

print('BLEU score without Attention: %f' % (np.mean(compute_BLEU(pure_seq2seq,
                                            test_data_loader,
                                            greedy_decode, trg_vocab_set))))
```

```
100%|██████████| 1139/1139 [00:24<00:00, 47.40it/s]BLEU score without Attention: 5.7921
```

Top k decode

```
test_set = MTDataset(test_src_sentences_list, src_vocab_set,
                     test_trg_sentences_list, trg_vocab_set, sampling=1.)
test_data_loader = data.DataLoader(test_set, batch_size=1, num_workers=4,
                                   shuffle=False)
```

```
                                          shuffle=False)

print('BLEU score without Attention: %f' % (np.mean(compute_BLEU(pure_seq2seq,
                                            test_data_loader,
                                            topk_decode, trg_vocab_set))))
```

```
  0%|          | 0/1139 [00:01<?, ?it/s]
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-21-ed03659f5738> in <module>()
      6 print('BLEU score without Attention: %f' % (np.mean(compute_BLEU(pure_seq2seq,
      7                                             test_data_loader,
----> 8                                             topk_decode, trg_vocab_set))))

1 frames

<ipython-input-19-71d92f78e908> in topk_decode(model, src_ids, src_lengths, max_len)
     13     top_k_proba = probabilities[topk_proba_indices]
     14     renormalized_proba = top_k_proba/top_k_proba.sum() # here, we have a
distribution over the different tokens: we will sample from it
---> 15     print('Renormalized proba', renormalized_proba.cpu())
     16     selected_token = np.random.choice(a=topk_proba_indices.numpy(), size=1,
p=renormalized_proba.numpy())
     17     outputs.append(selected_token)

RuntimeError: CUDA error: device-side assert triggered
```